Simulation auf Höchstleistungsrechnern

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 11/12

Organisatorisches

- Dozent: Stefan Lang, Paralleles Rechnen, IWR, Raum 532
- Veranstaltung: 4 V + 2 Ü
- Termine
  - Vorlesung: Di 9.00-11.00 (V), Do 9.00-11.00 (V)
  - Übungen: Mi 9.00-11.00 (Ü im CIP Pool, OMZ, INF 350 U.012)
- Voraussetzungen:
  Grundvorlesungen in Informatik und Numerik
- Hilfreich:
  Kenntnisse in C/C++

Was ist Wissenschaftliches Rechnen?

Im speziellen Numerische Simulation (NS):

- Ziel der NS ist natürliche oder technische Vorgänge auf Rechnern zu simulieren
- Interdisziplinärer Zugang: Naturwissenschaftler, Ingenieure, Mathematiker und Informatiker arbeiten zusammen
- Praktisch relevante Probleme werden mit formalen Methoden bearbeitet
- NS ermöglicht Erkenntnisgewinn in Bereichen die in Laborexperimenten und Feldstudien schwer zugänglich sind, z.B. Neurowissenschaften, Wasserwirtschaft, Astrophysik

Warum Wissenschaftliches Hochleistungsrechnen?

Trends in der **Numerischen Simulation**:

- Betrachtung von **Globalmodellen** statt lokalen Teilmodellen (virtuelle Prototypen im Bauingenieurwesen, Flugzeug-, Schiffs- und Automobilbau)
- Analyse von **gekoppelten Gesamtsystemen** statt isolierter Einzelprozesse (Mehrmedien-, Mehrphasen-, Mehrskalenprozesse, Konvektion-Diffusion-Reaktion)
- Rechner wird zum **wissenschaftlichen Beobachtungsinstrument** (hohes Auflösungsvermögen, Messungen in manchen Bereichen schwierig/unmöglich, Parameterstudien durchführbar)

Computational Neuroscience - Signalverarbeitung

- Ziel: Entwicklung eines Neuronennetzwerks, welches das beobachtete/gemessene Verhalten widerspiegelt.
- Simulation von Neuronennetzwerken, Statistische Analyse von Realisierungen

Ein Superrechner

ASCI Red Storm

Nachfolger des ersten TeraFlop Rechners ASCI Red 1997

Hardware:

- 11.646 $\times$ 2 GHz AMD Opteron CPUs
- CPU boards vertically mounted in 108 cabinets
- 4 GFLOPS per CPU (40 TFLOPS total)
- 1 GB per CPU (10 TB total)
- shared memory inside the node
- 3D mesh full interconnect

Software:

- compute nodes: custom Sandia-developed light-weight OS code-named Catamount
- service and storage nodes SuSE Linux.

Ein Superrechner

ASCI Red Storm

Skalierbarkeit

Algorithmische Komplexität am Beispiel von Lösern
für ein Gleichungssystem der Gestalt

$$Ax = b$$

Dimension	$d = 2$	$d = 3$
Gaussian elimination	$O(N^3)$	$O(N^3)$
Banded Gauss	$O(N^2)$	$O(N^{2.33})$
Nested Disection Gauss	$O(N^{1.5})$	$O(N^2)$
Richardson, GS, Jacobi	$O(N^2)$	$O(N^{1.67})$
CG, SOR	$O(N^{1.5})$	$O(N^{1.33})$
SSOR–CG	$O(N^{1.25})$	$O(N^{1.17})$
Multigrid	$O(N)$	$O(N)$
Cascadic Iteration	$O(N)$	$O(N)$

Skalierbarkeit

Skalierbarkeit bedeutet vereinfacht
Ein immer größeres Problem lässt sich auf einem immer größeren Computer stets gleichschnell berechnen.

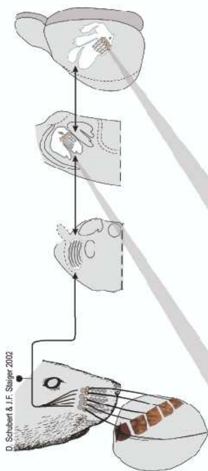
Dies ist selten der Fall (leider)!

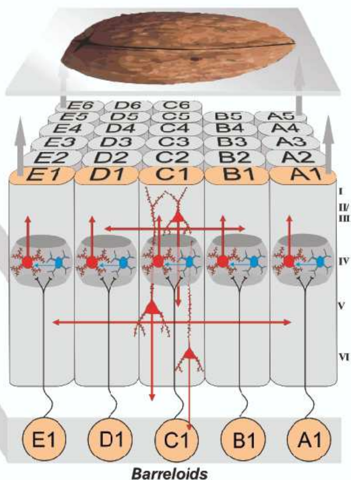
Konsequenz:

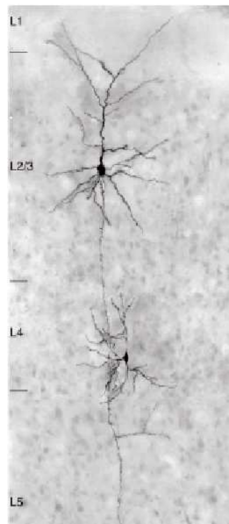
- leistungsfähige Software für leistungsfähige Rechner
- skalierbare Algorithmen + skalierbare Implementierungen + skalierbare Architekturen

. . . allein riesige Rechner zu kaufen reicht nicht aus!!

Ein biologisches System: Der Barrel Cortex der Ratte







Ziel: Mechanistisches Verständnis von einfachen Entscheidungsfindungsprozessen

Modellierung von Neuronaler Aktivität

Hodgkin-Huxley Gleichung:
Das elektr. Potential $v(\mathbf{x}, t)$ und Zustandspartikel $\mathbf{c}(\mathbf{x}, t) = (m(\mathbf{x}, t), h(\mathbf{x}, t), n(\mathbf{x}, t))^T$ obey

$$c_m \partial_t v = \partial_{\mathbf{x}} g_a \partial_{\mathbf{x}} v + i_{inj} - \sum_{\nu \in \mathcal{C}} i_\nu(v, \mathbf{c}) - i_s(v)$$

$$\partial_t c_\mu = \alpha_\mu(v) \cdot (1 - c_\mu) \; - \; \beta_\mu(v) \cdot c_\mu,$$

wobei $\nu \in \mathcal{C} =: \{Na, L, K\}$ and $\mu \in \{m, h, n\}$.
Rand- und Anfangsbedingungen sind gegeben durch

$$g_a \partial_{\mathbf{x}} v = g_N \qquad \text{on } \partial\Omega_N,$$

$$v = g_D \qquad \text{on } \partial\Omega_D,$$

$$(v, \mathbf{c})(\mathbf{x}, 0) = (v^0, \mathbf{c}^0) \text{ for } t = 0.$$

Die Ionenströme werden mit den Zustandpartikeln modelliert und sind gegeben durch

$$i_{Na}(v) = \overline{g_{Na}} \cdot m^3 h \cdot (v - E_{Na}),$$

$$i_K(v) = \overline{g_K} \cdot n^4 \cdot (v - E_K),$$

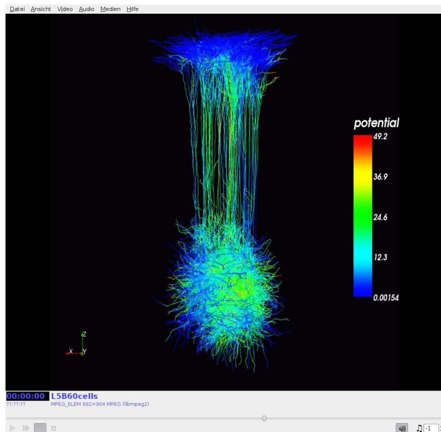
$$i_L(v) = \overline{g_L} \cdot (v - E_L)$$

Darüberhinaus werden Ströme von Synapsen modelliert durch

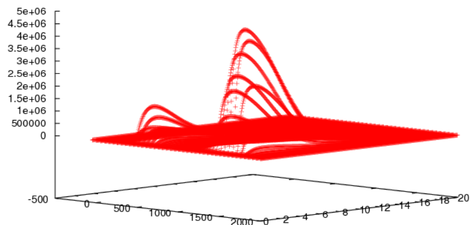
$$i_s(v, t) = g_s(t) \cdot (v - E_s)$$

mit zeitabhängiger synaptischer Stärke g_s.

Aktuelle Arbeit: passive Auslenkung des Barthaars

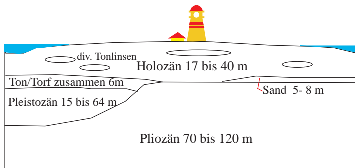


60 L5B Neuronen aktiviert vom VPM



Räumliche und zeitliche Aktivitätsverteilung

Problemfall Norderney



- Projekt "Küstenschutz"
- Inseltypische Süßwasserlinse (-85m)
- Simulation des Linsenaufbaus und der Wasserförderung aus zwei Pumpen
- Strategie zur Erhaltung der Wasserqualität

Problemfall Norderney

10km x 4km
x 150m

- Anfangsgitter (1516 elements, D. Feuchter, geomod2ng)
- 6 geologische Schichten mit variierender Permeabilität $10^{-10} - 10^{-15}$
- Randbedingungen: Einfluß von Süßwasser am oberen Rand Quellen sind Senken, Ein/Ausfluß in Küstenbereichen, hydrostatischer Druck

Dichtegetriebene Grundwasserströmung

Gleichungen der dichtegetriebenen Grundwasserströmung abgeleitet aus Erhaltungssätzen.
Formulierung verwendet **Salzmassenbruch** ω und **Druck** p

$$\begin{aligned} \partial_t(n\rho) + \nabla \cdot (\rho \mathbf{v}) &= Q\rho, & \text{(f low)} \\ \partial_t(n\rho\omega) + \nabla \cdot (\rho \mathbf{v}\omega - \rho D \nabla \omega) &= Q\rho\omega & \text{(transp.)} \end{aligned}$$

mit

$$\begin{aligned} \mathbf{v} &= -K/\mu(\nabla p - \rho \mathbf{g}), & \text{(Darcy's law)} \\ D &= (\alpha_L - \alpha_T)\mathbf{v}/|\mathbf{v}| + \alpha_T|\mathbf{v}| & \text{(Scheidegger)} \end{aligned}$$

Passende Anfangs- und Randwerte müssen definiert werden.

Methoden- und Softwareentwicklung

Zusätzlich sind realistische Probleme schwer zu bearbeiten

All dies zusammen zu realisieren ist eine schwierige Aufgabe!
Softwareengineering nicht Gegenstand der Vorlesung (Prof. Ludewig)

TOP500 - Liste der SuperComputer

TOP500

TOP500 - Performance

1 Pflop/s
100 Tflop/s
10 Tflop/s
1 Tflop/s
100 Gflop/s
10 Gflop/s
1 Gflop/s
100 Mflop/s

SUM
N=1
N=500
220 TF/s
35.8 TF/s
1.17 TF/s
59.7 GF/s
0.4 GF/s
134 GF/s
Fujitsu 'NWT' NAL
Intel ASCI Red Sandia
IBM ASCI White LLNL
NEC Earth Simulator
My Laptop

Jun-93 Nov-93 Jun-94 Nov-94 Jun-95 Nov-95 Jun-96 Nov-96 Jun-97 Nov-97 Jun-98 Nov-98 Jun-99 Nov-99 Jun-00 Nov-00 Jun-01 Nov-01 Jun-02

TOP500 - Entwicklung

TOP500

Performance Extrapolation

10 PFlop/s
1 PFlop/s
100 TFlop/s
10 TFlop/s
1 TFlop/s
100 GFlop/s
10 GFlop/s
1 GFlop/s
100 MFlop/s

Sum
N=1
N=500
Earth Simulator
ASCI Purple

Jun-93 Jun-94 Jun-95 Jun-96 Jun-97 Jun-98 Jun-99 Jun-00 Jun-01 Jun-02 Jun-03 Jun-04 Jun-05 Jun-06 Jun-07 Jun-08 Jun-09 Jun-10

Petaflop Rechner in 2010! Exaflop bis 2015?

Parallelisierung: Ein Einführungsbeispiel

Skalarprodukt zweier Vektoren

- Einführung einer geeigneten Notation
- Interaktion über gemeinsame Variable
- Interaktion über Nachrichten
- Bewertung paralleler Algorithmen

Ein einfaches Problem: Skalarprodukt bilden

Skalarprodukt zweier Vektoren der Länge N:

$$s = x \cdot y = \sum_{i=0}^{N-1} x_i y_i.$$

Parallelisierungsidee:

1. Summanden $x_i y_i$ unabhängig
2. $N \geq P$, bilde $I_p \subseteq \{0, \dots, N-1\}$, $I_p \cap I_q = \emptyset \; \forall p \neq q$
   Jeder Prozessor berechnet dann die Teilsumme $s_p = \sum_{i \in I_p} x_i y_i$
3. Summation der Teilsummen z.B. für $P = 8$:

$$s = \underbrace{\underbrace{\underbrace{s_0 + s_1}_{s_{01}} + \underbrace{s_2 + s_3}_{s_{23}}}_{s_{0123}} + \underbrace{\underbrace{s_4 + s_5}_{s_{45}} + \underbrace{s_6 + s_7}_{s_{67}}}_{s_{4567}}}_{s}$$

Grundlegende Begriffsbildungen

Sequentielles Programm: Folge von Anweisungen die der Reihe nach abgearbeitet werden.

Sequentieller Prozess: Aktive Ausführung eines sequentiellen Programmes.

Parallele Berechnung: Menge interagierender sequentieller Prozesse.

Paralleles Programm: Beschreibt parallele Berechnung. Gegeben durch Menge von sequentiellen Programmen.

Notation für parallele Programme

- Möglichst einfach, losgelöst von praktischen Details
- Erlaubt verschiedene Programmiermodelle

Programm (Muster eines parallelen Programmes)

```
parallel <Programmname>
{
    // Sektion mit globalen Variablen (von allen Prozessen zugreifbar)
    process <Prozessname-1> [<Kopienparameter>]
    {
        // lokale Variablen, die nur von Prozess <Prozessname-1>
        // gelesen und geschrieben werden können
        // Anwendungen in C-ähnlicher Syntax. Mathematische
        // Formeln oder Prosa zur Vereinfachung erlaubt.
    }
    ...
    process <Prozessname-n> [<Kopienparameter>]
    {
        ...
    }
}
```

Notation für parallele Programme II

Variablendeklaration

double x, $y[P]$;

Initialisieren

int $n[P] = \{1[P]\}$;

lokale/globale Variablen
Bemerkungen zum Prozessbegriff

Skalarprodukt mit zwei Prozessen

Programm (Skalarprodukt mit zwei Prozessen)

```
parallel two-process-scalar-product
{
    const int N=8;                       // Problemgröße
    double x[N], y[N], s=0;              // Vektoren, Resultat
    process Π1
    {
        int i;
        double ss=0;
        for (i = 0; i < N/2; i++)
            ss += x[i]*y[i];
        s=s+ss;                          // Gefahr!
    }
    process Π2
    {
        int i;
        double ss=0;
        for (i = N/2; i < N; i++)
            ss += x[i]*y[i];
        s=s+ss;                          // Gefahr!
    }
}
```

- Variablen sind global, jeder Prozess bearbeitet Teil der Indizes
- Kollision bei schreibendem Zugriff!

Kritischer Abschnitt I

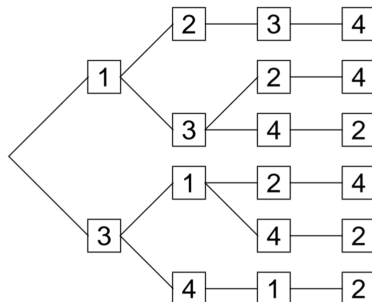
- Hochsprachenanweisung $s = s + ss$ wird zerlegt in Maschinenbefehle:

	Prozess Π_1		Prozess Π_2
1	lade s in R1	3	lade s in R1
	lade ss in R2		lade ss in R2
	add R1 und R2 Ergebnis in R3		add R1 und R2 Ergebnis in R3
2	speichere R3 nach s	4	speichere R3 nach s

- Ausführungsreihenfolge der Anweisungen verschiedener Prozesse ist nicht festgelegt.

Kritischer Abschnitt II

- Mögliche Ausführungsreihenfolgen sind:



Ergebnis der Berechnung

$s = ss_{\Pi_1} + ss_{\Pi_2}$

$s = ss_{\Pi_2}$

$s = ss_{\Pi_1}$

$s = ss_{\Pi_2}$

$s = ss_{\Pi_1}$

$s = ss_{\Pi_1} + ss_{\Pi_2}$

- Nur die Reihenfolgen 1-2-3-4 oder 3-4-1-2 sind korrekt.

Kritischer Abschnitt III

- Anweisungsblock bildet *kritischen Abschnitt*, der unter *wechselseitigem Ausschluß* bearbeitet werden muss.
- Wir notieren dies *zunächst* mit eckigen Klammern

$$[\ \langle\text{Anweisung 1}\rangle; \ldots; \langle\text{Anweisung } n\rangle;\ ]$$

- Das Symbol „[“ wählt einen Prozess aus der den kritischen Abschnitt bearbeiten darf, alle anderen warten.
- Effiziente Umsetzung erfordert Hardwareinstruktionen, die wir später besprechen.

Parametrisieren von Prozessen

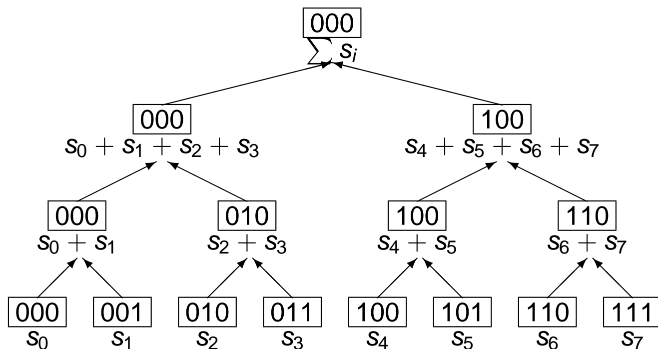
- Prozesse enthalten teils identischen Code (auf unterschiedlichen Daten)
- Parametrisiere den Code mit einer Prozessnummer, wähle zu bearbeitende Daten mit dieser Nummer aus
- SPMD = single program multiple data

Programm (Skalarprodukt mit P Prozessoren)

```
parallel many-process-scalar-product
{
    const int N;                    // Problemgröße
    const int P;                    // Anzahl Prozesse
    double x[N], y[N];              // Vektoren
    double s = 0;                   // Resultat
    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i; double ss = 0;
        for (i = N ∗ p/P; i < N ∗ (p + 1)/P; i++)
            ss += x[i]*y[i];
        [s = s + ss];               // Hier warten dann doch wieder alle
    }
}
```

Kommunikation in hierarchischer Struktur

Baumartige Organisation der Kommunikationsabfolge mit ld P Stufen



In Stufe $i = 0, 1, \ldots$

- Prozesse, deren letzte $i + 1$ Bits 0 sind, holen
- Ergebnisse von den Prozessoren deren letzte i Bits 0 und deren Bit i 1 ist

```
parallel parallel-sum-scalar-product
{
    const int d = 4;
    const int N = 100;                          // Problemgröße
    const int P = 2^d;                          // Anzahl Prozesse
    double x[N], y[N];                          // Vektoren
    double s[P] = {0[P]};                       // Resultat
    int flag[P] = {0[P]};                       // Prozess p ist fertig

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i, r, m, k;

        for (i = N ∗ p/P; i < N ∗ (p + 1)/P; i++)
            s[p]+ = x[i] ∗ y[i];

        for (i = 0; i < d; i++)
        {
            r = p & [∼ (∑_{k=0}^{i} 2^k)];      // lösche letzten i + 1 bits
            m = r | 2^i;                        // setze Bit i
            if (p == m) flag[m]=1;
            if(p == r)
            {
                while(!flag[m]);                // Bedingungssynchronisation
                s[p] = s[p] + s[m];
            }
        }
    }
}
```

Parallelisieren der Summe II

- Neue globale Variablen: $s[P]$ Teilergebnisse $flag[P]$ zeigt an, dass Prozessor fertig ist
- Das Warten nennt man *Bedingungssynchronisation*
- In diesem Beispiel konnte wechselseitiger Ausschluß durch eine Bedingungssynchronisation ersetzt werden. Dies geht nicht immer!
- Liegt daran, dass wir hier die Reihenfolge vorab festgelegt haben

Lokalisieren

Ziel: Vermeiden globaler Variablen

Wir gehen zweistufig vor: (I) Mache Vektoren x, y lokal, (II) mache Ergebnis s lokal

Programm (Skalarprodukt mit lokalen Daten)

```
parallel local-data-scalar-product
{
    const int P, N;
    double s = 0;

    process Π [ int p ∈ {0, . . . , P − 1}]
    {
        double x[N/P], y[N/P]; // Annahme N durch P teilbar
                                // Lokaler Ausschnitt der Vektoren
        int i;
        double ss=0;

        for (i = 0,i < (p + 1) ∗ N/P − p ∗ N/P;i++) ss = ss + x[i] ∗ y[i];
        [s = s + ss; ]
    }
}
```

Jeder speichert nur N/P Indizes (einer mehr falls nicht exakt teilbar), *diese beginnen immer mit der lokalen Nummer 0*

Jeder lokale Index entspricht einem globalen Index im sequentiellen Programm:

$$i_{\text{global}}(p) = i_{\text{lokal}} + p * N/P$$

Nachrichtenaustausch I

Um völlig auf globale Variablen verzichten zu können brauchen wir ein neues Konzept: *Nachrichten*

Syntax:

send($<$Process$>$,$<$Variable$>$)
receive($<$Process$>$,$<$Variable$>$)

Semantik: **send** schickt den Inhalt der Variablen an den angegebenen Prozess, **receive** wartet auf Nachricht von dem angegebenen Prozess und füllt diese in die Variable

send wartet bis die Nachricht erfolgreich empfangen wurde, **receive** blockiert den Prozess bis eine Nachricht empfangen wurde

Blockierende, oder synchrone Kommunikation (später andere)

Nachrichtenaustausch II

Programm (Skalarprodukt mit Nachrichtenaustausch)

parallel *message-passing-scalar-product*
{
  const int d, $P = 2^d$, N; *// Konstanten!*

  process Π [**int** $p \in \{0, \ldots, P-1\}$]
  {
    double $x[N/P]$, $y[N/P]$; *// Lokaler Ausschnitt der Vektoren*
    int i, r, m;
    double s, $ss = 0$;

    for $(i = 0, i < (p+1) * N/P - p * N/P; i{+}{+})$ $s = s + x[i] * y[i]$;
    for $(i = 0, i < d, i{+}{+})$ *// d Schritte*
    {
      $r = p \,\&\, \left[\sim \left(\sum_{k=0}^{i} 2^k\right)\right]$;
      $m = r \mid 2^i$;
      if $(p == m)$
        send(Π_r, s);
      if $(p == r)$
      {
        receive(Π_m, ss);
        $s = s + ss$;
      }
    }
  }
}

Bewertung paralleler Algorithmen I

Hier: asymptotisches Verhalten in Abhängigkeit der Problemgröße und Prozessorzahl

Sequentielle Laufzeit:

$$T_s(N) = 2Nt_a,$$

t_a: Zeit für arithmetische Operation

Parallele Laufzeit für message-passing Variante:

$$T_p(N,P) = \underbrace{2\frac{N}{P}t_a}_{\text{lokales Skalarprodukt}} + \underbrace{\operatorname{ld} P(t_m + t_a)}_{\text{parallele Summe}},$$

t_m: Zeit für Senden einer Zahl

Speedup:

$$S(N,P) = \frac{T_s(N)}{T_p(N,P)} = \frac{2Nt_a}{2\frac{N}{P}t_a + \operatorname{ld} P(t_m + t_a)} = \frac{P}{1 + \frac{P}{N}\operatorname{ld} P\frac{t_m+t_a}{2t_a}}$$

Es gilt $S(N,P) \leq P$!

Bewertung paralleler Algorithmen II

Effizienz:

$$E(N,P) = \frac{S(N,P)}{P} = \frac{1}{1 + \frac{P}{N} \operatorname{ld} P \frac{t_m + t_a}{2t_a}}$$

Es gilt $E \leq 1$

asymptotische Aussagen:

- festes N, wachsendes P: $\lim_{P \to \infty} E(N,P) = 0$
- festes P, wachsendes N: $\lim_{N \to \infty} E(N,P) = 1$
  Für welches Verhältnis $\frac{P}{N}$ „akzeptable" Effizienzwerte erreicht werden, regelt der Faktor $\frac{t_m + t_a}{t_a}$, das Verhältnis von Kommunikations- zu Rechenzeit.
- Skalierbarkeit bei gleichzeitigem Anwachsen von N und P in der Form $N = kP$:

  $$E(kP,P) = \frac{1}{1 + \operatorname{ld} P \frac{t_m + t_a}{2t_a k}}$$

  Fällt sehr langsam mit P ab $\rightarrow$ gut skalierbar.

Beispielhaft für viele Algorithmen

Themen

Hardware

- (Parallele) Rechnerarchitekturen: SMP, Vektorrechner, Parallelrechner
- Realisierungen: einige Architekturen im Detail (Paragon, ASCI Red Storm, IBM Blue Gene)

Programmiermodelle

- Programmiermodelle: OpenMP, MPI, PThreads
- Grundlegende Numerische Algorithmen: Matrixmultiplikation

Algorithmen

- Leistungsbewertung: Effizienz, Beschleunigung, Skalierbarkeit
- Paralleles Sortieren
- Dicht- und dünnbesetzte Gleichungssysteme

Anwendungen

- Parallele (Numerische) Anwendungen (Neurowissenschaften, Biologie, Bodenphysik, Astrophysik)

Die Themen sind unverbindlich.

Organisatorisches

- Morgen, Mi 11.00-13.00 sind Übungen im IWR, 532
  sonst: Mi, 9.00-11.00, CIP Pool, OMZ, INF 350, U.012
- Dan Popovich: Einführung in C++
- Scheinvergabe: Regelmässige Teilnahme, 50% Punkte
- Prüfungen: nach Rücksprache