Übersicht:

- **Performance Optimierungen** („gutes“ C++, Wissen um die **Hardware Architektur** [Caches], **Expression Templates** für effizientes Operator-Überladen und „schönen“ Code)
- **Parallelisierung**: OpenMP (gemeinsamer Speicher) und MPI (verteilter Speicher)
- **Partielle Differentialgleichungen** → Diskretisierung mit **Finiten Differenzen**
- **Lösen linearer Gleichungssysteme**:
    - **Direkte Löser**: Gauss-Elimination (LR-Zerlegung) [$O(n^3)$]
    - **Iterative Löser**:
        - Jacobi, (red-black) Gauss-Seidel, SOR
        - (quadratische Form), Steepest Descent, CD, CG [Krylov-Unterraum-Verfahren]
        - GMRES [Krylov-Unterraum-Verfahren] → Vorkonditionierung !
- **Gewöhnliche Differentialgleichungen**

Simulationen brauchen:

- Viel Rechenleistung → Flops (Mega=6, Giga=9, Tera=12, Peta=15)
- Viel Speicher → Bytes
- Moore’s Law: Verdopplung der Transistoren alle 18 Monate (Takt nur langsam bzw. stagnierend)

Parallelität:

- Superskalarität (parallel arbeitende Funktionseinheiten → parallele Ausführung mehrerer Instruktionen) [→ out-of-order execution, branch-prediction/speculative execution]
- Pipelining (Aufbrechen in abhängige Stufen [ein „Clock-Cycle“ pro „Stufe“]: fetch/load → decode → reserve registers → execute → … → write back result) [→ out-of-order, branch-prediction]
- Mehrere Kerne (Multicore)
- Mehrere CPUs auf einem Board (→ gemeinsamer Speicher)
- Verteilter Speicher: Cluster Computing (schnelle Vernetzung) & Grid Computing (Internet)

Speicher-Wand (Memory-Wall – von-Neumann Bottleneck):

- Bandbreite (Busbreite x Takt) und … [→ ebenso Bandbreite & Latenz des Netzwerks]
- … Latenz (viele Nanosekunden) halten nicht mit CPU (1Ghz → 1 ns) mit→ Register, Caches
- Memory Interleaving: Aufeinanderfolgende Array-Elemente werden verschiedenen Speicherbänken zugeteilt, auf die parallel zugegriffen wird (→ verbessert Bandbreite/Durchsatz, nicht Latenz)

Der beste Algorithmus für eine Simulation zeichnet sich aus durch:

- Das beste Model für das Problem
- Die beste Diskretisierung (Adaptivität!)
- Den besten Löser
- Die beste Implementierung (Ausnutzung der Hardware [→ Parallelität!])
- Gute Bibliotheken! (teilweise extrem optimiert für bestimmte Hardware)

Caches:

- Hält Kopien aus dem RAM
- Niedrige Latenz und hohe Bandbreite
- Cache Hit = Speicher liegt im Cache – Cache Miss = Speicher muss aus RAM angefordert werden
- Problem: Konsistenz mit RAM = Cache Kohärenz
- $Speedup = \frac{Time_{Memory}}{H \cdot Time_{Cache} + (1-H) \cdot Time_{Memory}}$ → erst mit hoher Hit-Rate H lohnt sich der Cache
- Assoziativität:
    - Direct mapped: jedes Speicherwort kann nur an genau eine Stelle im Cache (→ schnell)
    - Fully associative: jedes Speicherwort kann überall hin im Cache (→ teuer)

    - Set associative: jedes Speicherwort kann an eine von k Stellen im Cache (→ Kompromiss)
- Write-back (dirty bit) vs. Write-Through
- Ersetzungs-Strategie (z.B LRU)
- Größe einer Cache-Line?
- Cache-Misses:
    - Compulsory: erster Zugriff auf eine Speicherstelle
    - Capacity: Cache ist voll, Daten werden überschrieben und fliegen wieder raus
    - Conflict: zwei Speicherwörter sind dem gleichen Cache-Slot zugeordnet (direct & set-ass.)

Code Optimieren/Beschleunigen:

- Schnelle perfekt optimierte Bibliotheken verwenden
- Compiler: den „richtigen“ mit den „richtigen“ Compiler-Flags wählen
- Array-Layout bei iterativem Zugriff (→ „data-locality”)
- Gemeinsame algebraische Unterausdrücke temporär vorberechnen (Assoziativität? Seiteneffekte?)
- Schleifen-Invarianten temporär vorberechnen
- Spezialbefehle der CPU verwenden (MMX, SSE, EM64T etc. → z.B. Multiply-Add)
    - SIMD (klassische Vektormaschinen, CUDA, …)
    - Speicher-Alignment → Vectorization
    - Intel Compiler→ pragmas
    - Richtig Effizient: gleich selber in Assembler schreiben (ansonsten zumindest den vom Compiler erzeugten Assembler-Code kontrollieren)
- Instruction-Level-Parallelism
- Loop-Unrolling (Evtl. Sonderbehandlung für das Schleifenende nötig) → Aufpassen, dass Register nicht „ausgehen“ [kleine Schleifen evtl. komplett auflösen]
- Software-Pipelining (nicht so wichtig bei out-of-order Prozessoren) [„Software-Prefetching“]
- Aliasing (verboten in Fortran → C++ muss „vorsichtiger“ vorgehen und „konservativen“ Code erzeugen)
- Funktionsaufrufe sind teuer (bis zu 100 CPU Zyklen) → Inline
- Data-Locality:
    - Temporal Locality: Daten passen in den Cache und werden häufig daraus verwendet (schwer zu erreichen)
    - Spatial Locality: Alle Daten, die in den Cache geladen werden, werden mind. einmal benutzt → Cache-Miss teilt sich über die ganze Cache-Line auf (durch geschickte Datenstrukturen oft zu erreichen)
        - Square-Blocking! (auch das machen Compiler, sofern die Schleifen nicht zu komplex sind) [Vedoppelt Anzahl der Schleifen, kann sehr kompliziert werden]
- Zwei Schleifen (die [unter anderem] über die gleichen Daten laufen) zu einer zusammenfassen
- Cache-Thrashing: Problem bei Conflict-Misses → Lösung: Dummy Elemente einbauen, damit Speicherwörter verschiedenen Cache-Slots zugeordnet werden („böse“: 2^X Array-Dimensionen)
- IF-Anweisungen vor allem im Inneren von Schleifen vermeiden (→ Schleifen aufspalten)
- Prefetching (mit anderen Berechnungen möglichst Speicher-Ladezeit überdecken)
- Compile-Zeit Auswertung von Funktionen → Expression Templates

Expression Templates:

- „Schönes Software-Design“ vs. Effizienz
- Operator-Überladen: Auswertung erst mit dem „=“-Operator durch Aufbauen eines Expression-Objekts auf der rechten Seite (alles zu Compile-Zeit!) [kein temporäres Allokieren von Objekten!]
- Operationen wie +/-/*/etc. auf Expression-Objekte geben ein Expression-Objekt zurück (dieses bietet wie alle Expression-Objekte „[]“ als Zugriff an) [kein temporäres Allokieren von Objekten!]
- Elementar wichtig, damit Compile-Zeit Auswertung funktioniert: inline!
- Ermöglicht einfache Parallelisierung mit OpenMP

Finite Differenzen:

- $\mathrm{grad(u)} = \nabla u(x,y) = (\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y})$
- $\Delta u(x,y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = (\nabla \cdot \nabla) u(x,y)$
- Differenzenquotienten:
  - $\frac{\partial u}{\partial x} = \frac{u(x+h)-u(x)}{h}$ (vorwärts) $\rightarrow \frac{1}{h}(0\ -1\ 1)$ [Konsistenz-Ordnung: 1 – O(h)]
  - $\frac{\partial u}{\partial x} = \frac{u(x)-u(x-h)}{h}$ (rückwärts) $\rightarrow \frac{1}{h}(-1\ 1\ 0)$ [Konsistenz-Ordnung: 1 – O(h)]
  - $\frac{\partial u}{\partial x} = \frac{u(x+h)-u(x-h)}{2h}$ (zentral) $\rightarrow \frac{1}{2h}(-1\ 0\ 1)$ [Konsistenz-Ordnung: 2 – O(h^2)]
  - $\frac{\partial^2 u}{\partial x^2} = \frac{u(x-h)-2u(x)+u(x+h)}{h^2} \rightarrow \frac{1}{h^2}(1\ -2\ 1)$ [Konsistenz-Ordnung: 2 – O(h^2)]
- Taylor-Entwicklung: $f(x+h) = f(x) + \frac{h}{1!}f'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \frac{h^4}{4!}f''''(x) + O(h^5)$
- Vergleich analytische/exakte Lösung mit nummerischer Lösung: Diskretisierungsfehler (durch die Differenzenquotienten) und algebraische Fehler (durch den [iterativen] Löser)!
- Stabilität: wenn das Ergebnis nur von den initialen Werten abhängt
- Theorem: wenn eine Diskretisierung stabil und konsistent mit der Ordnung p ist, dann konvergiert die numerische Lösung mit Ordnung p

Direkte Löser – Gauss-Elimination:

- Spaltenweise Elimination eines Zeilenelements (zusammen mit rechter Seite) [Pivotisierung]
- LR Zerlegung für mehrere rechte Seiten (→ Vorwärts- & Rückwärts-Substitution)
- Es gibt eine eindeutige Lösung, wenn die Matrix nicht-singulär ist
- Exakte Lösung (Rundungsfehler) nach einer endlichen Anzahl an Schritten
- Voll-besetzte Matrix → O(n^3)
- X-Diagonal-Matrix → O(n) ! (z.B. bei 1-dimensionalen gewöhnlichen Differentialgleichungen)
- Dünn-besetzte Matrizen
- Band-Struktur der Matrix bei strukturiertem symmetrischen Grid
- Allgemeine Steifigkeitsmatrizen bei FE: unstrukturiert
- Elimination: existierende Nullen werden wieder zerstört → „fill-in“
- „Gute“ Nummerierung finden (2D: findet praktisch Anwendung – 3D: Elimination ist nicht effizient genug)
- Direkt: QR, Cholesky Zerlegung

Iterative Löser:

- Grundidee: Sequenz von approximierten Lösungsvektoren, die hin zur Lösung konvergieren (dabei sollte die Berechnung eines Iterationsschritts „billig“ sein und die Konvergenz „schnell“ gehen)
- Gauss-Seidel [mit SOR → $u^{k+1} = (1-\omega)u^k + \omega(b - \dots)$]
- $x^{k+1} = x^k + B^{-1}\left(b - Ax^k\right)$
- $\left\|e^{k+1}\right\| \leq \|Id - B^{-1}A\|\left\|e^k\right\| \rightarrow \|Id - B^{-1}A\| \leq 1$
- $A = L + D + R$:
  - $A = \hat{A} + E \rightarrow \hat{A}x^{k+1} = b - Ex^k$
  - Jacobi:
    - $x^{k+1} = D^{-1}(b - (L+R)x^k)$
    - $x^{k+1} = x^k + D^{-1}(b - Ax^k)$
  - Gauss-Seidel:
    - $x^{k+1} = (L+D)^{-1}(b - Rx^k)$
    - $x^{k+1} = x^k + (L+D)^{-1}(b - Ax^k)$
  - Gauss-Seidel mir SOR: $x^{k+1} = x^k + (L + \frac{1}{w}D)^{-1}(b - Ax^k)$
- Elliptische selbst-adjungierte partielle Differentialgleichungen führen zu symmetrischen positiv-definiten FE Steifigkeits-Matrizen (Gauss-Seidel konvergiert immer)

    - Speziell bei finiten Differenzen: Gauss-Seidel konvergiert immer, wenn A strikt diagonaldominant oder diagonaldominant und nicht-singulär ist
- Speicherung der Matrix:
    - Nur den Stencil
    - Band-Matrix: Jedes Band in einem 1D-Vektor
    - Unstrukturiert und dünn besetzt: CRS (jede Zeile Wert+Index)
- Abbruch-Kriterium der Iteration:
    - Residuum $r_k = b - Ax_k \quad \rightarrow \quad r_k = Ae_k$
    - Auch wenn $\|r_k\|$ klein ist heißt das nicht, dass auch $\|e_k\|$ klein ist (hängt von der Matrix ab!)

Die quadratische Form:
- Positiv-definit: $x^T Ax > 0 \; \forall x \neq 0 \; \rightarrow$ alle Eigenwerte sind positiv!
- Quadratische Form: $f(x) = \frac{1}{2}x^T Ax - b^T x + c$
- $f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b$
- A ist symmetrisch: $f'(x) = Ax - b$
- A ist pos.-def.: f(x) hat die Form einer parabolischen Schüssel
- $Ax = b$ kann gelöst werden, indem das Minimum von f(x) gesucht wird

Steepest Descent:
- Residuum: $r_k = b - Ax_k$
- Fehler: $r_k = -Ae_k$ mit $e_k = x_k - x$
- Schritt in die steilste Richtung (= Residuum): $-f'(x_k) = b - Ax_k = r_k$
- Wie weit? Schnitt entlang der Richtung führt zu einer Parabel → wie weit bis zu diesem Minimum?
  An diesem Punkt ist der Gradient orthogonal zum jetzigen! → $r_{k+1}^T r_k = 0 \; \rightarrow \; \alpha_k = \frac{r_k^T r_k}{r_k^T A r_k}$
- Iterations-Schritt: $x_{k+1} = x_k + \alpha_k r_k$
- Wenn Fehler ein Eigenvektor von A, dann Konvergenz in einem Schritt:
    - $r_k = -Ae_k = -\lambda_e e_k$
    - $e_{k+1} = e_k + \alpha_k r_k = 0$
- Allgemeiner:
    - Fehler ausgedrückt als die Summe aus den Eigenvektoren
    - Auch Residuum lässt sich dann als Summe der Eigenvektoren ausdrücken
    - Wenn alle Eigenwerte identisch sind: wieder Konvergenz in einem Schritt
- Steepest Descent und CG: im Gegensatz zu Jacobi verringern sie nicht in jedem Schritt jede Eigenvektor-Komponente. Manchmal kann eine einzige auch größer werden (euklidische Norm!).
- Energy-Norm: $\|e\|_A = \sqrt{(e^T Ae)}$
- Minimieren von $f(x_k)$ ist äquivalent mit dem Minimieren von $\|e_k\|_A$
- $\|e_k\|_A \leq (\frac{\kappa-1}{\kappa+1})^k \|e_0\|_A \quad mit \quad \kappa = \frac{\lambda_{max}}{\lambda_{min}} \quad \rightarrow \quad \|e_{k+1}\|_A \leq \|e_k\|_A$
- Unbestimmte Anzahl an Iterationsschritten bis zur Konvergenz!

Conjugate Directions:
- Idee: Schritt in eine Richtung nur noch einmal:
    - alle Suchrichtungen senkrecht zueinander
    - die Suchrichtung in Schritt k steht Senkrecht zum Fehler von Schritt k+1
    - Man wäre nach n Schritten bei $\vec{x} \in \mathbb{R}^n$ fertig (exakte Lösung!)
    - Problem: Fehler ist nicht bekannt
- Lösung:
    - Die Suchrichtungen sollen A-orthogonal zueinander sein: $d_i^T A d_j = 0 \; \forall i \neq j$
    - D.h. die Suchrichtung in Schritt k steht A-Orthogonal zum Fehler von Schritt k+1

- Das ist äquivalent mit dem finden des Minimums entlang der Suchrichtung d
- Ebenfalls fertig nach n Schritten

- Erzeugen der A-orthogonalen Suchrichtungen:
  - Gram-Schmidt (Benutzen der Koordinaten-Achsen um die Suchrichtungen zu erzeugen)
  - Um alle Suchrichtungen zu Berechnen: $O(n^3)$ Operationen (Gauss-Elimination!)
  - Alle Suchrichtungen müssen gespeichert werden
- Findet immer die beste Lösung im Unterraum $span\{d_0, d_1, \dots, d_{k-1}\}$, d.h. $\|e_k\|_A$ minimal – $e_k = e_0 + span\{d_0, d_1, \dots, d_{k-1}\}$

Conjugate Gradient (Verbesserung der Erzeugung der Suchrichtungen):
- Suchrichtungen werden aus dem Residuum erzeugt (nicht aus den Koordianten-Achsen)
- Residuum ist orthogonal zur den vorherigen Suchrichtungen und damit auch zu jedem vorherigen Residuum: $r_i^T r_j = 0 \; \forall i \neq j$
- r_i ist A-orthogonal zu allen d_j mit j < i-1
- Um d_k A-orthogonal zu allen vorherigen Suchrichtungen zu machen genügt d_{k-1}
- $d_k = r_k + \sum_{i=0}^{k-1} \beta_{ki} d_i$ reduziert sich damit zu $d_k = r_k + \beta_{k-1} d_{k-1}$
- Der Suchraum ist der Unterraum $span\{r_0, r_1, \dots, r_{k-1}\} = span\{d_0, Ad_0, A^2 d_0, \dots, A^{k-1} d_0\} = span\{r_0, Ar_0, A^2 r_0, \dots, A^{k-1} r_0\}$ [Krylov Unterraum], dort wird die beste Lösung gefunden
- $e_k = e_0 + span\{d_0, d_1, , \dots, d_{k-1}\} = e_0 + span\{r_0, Ar_0, A^2 r_0, \dots, A^{k-1} r_0\} = e_0 + span\{Ae_0, A^2 e_0, A^3 e_0, \dots, A^k e_0\}$
- Der Fehler e_k wird in der Energy-Norm minimiert, d.h. $\|e_k\|_A$ minimal
- Konvergenz nach n Schritten (Achtung: Rundungsfehler! Deswegen Residuum nach einigen Iterationen explizit wieder berechnen mit $r_k = b - Ax_k$)
- Konvergenzanalyse: $e_k = P_k(A)e_0$, …. , $\|e_k\|_A \leq 2(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})^k \|e_0\|_A$
- Vorkonditionierung:
  - Komplexität von CG hängt von der Konditionszahl ab: $\kappa = \frac{\lambda_{max}}{\lambda_{min}} \geq 1$
  - Idee: Die Konditionszahl verbessern
  - Löse $M^{-1}Ax = M^{-1}b$ mit $\kappa(M^{-1}A) \ll \kappa(A)$
  - Z.B.: M ist der Diagonalteil von A
  - SSOR, Multigrid,…

	SD	CD	CG
Suchrichtung	Residuum	A-orthogonale Richtungen	A-orthogonale Richtungen
Konstruktion der Suchrichtungen	-	Koordinaten-Achsen	Aus den Residuen
Konvergenz	Unendlich viele Schritte	Höchstens n Schritte	Höchstens n Schritte
Komplexität	$O(\kappa n)$		$O(\sqrt{\kappa} n)$
	Schlechte Konvergenz	Suchrichtungen müssen gespeichert werden, rechenintensiv	

Typische Komplexitäten bei Finite Differenzen und Finite Elemente Approximationen im 2D Fall:
- Gauss-Seidel: $O(n^2)$
- SOR: $O(n^{1.5})$
- Steepest-Descent: $O(n^2)$
- CG: $O(n^{1.5})$
- CG-SSOR: $O(n^{1.25})$
- CG-MG: O(n)

GMRES:

- Krylov-Unterraum-Verfahren: n-ter Krylov-Raum: $K_n = span\{b, Ab, A^2b, \dots, A^{n-1}b\}$
- GMRES findet die Lösung $x_n \in K_n$, für die $\|r_n\|_2 = \|b - Ax_n\|_2$ minimal ist (euklidische Norm!)
- Arnoldi Iteration findet orthogonale Vektoren $q_1, q_2, \dots, q_n$ die eine Basis für K_n bilden
- Mit exakter Arithmetik Konvergenz nach höchstens n-Schritten – es gilt: $\|r_{n+1}\|_2 \leq \|r_n\|_2$
- Geeignet auch für nicht-symmetrische Matrizen!
- Im worst-case kann GMRES beliebig langsam konvergieren
- Vorkonditionierung!

Parallelität in SD, CD, CG und GMRES:

- Sämtliche Vektor Operationen (trivial: zeilenweise unabhängig)
- Matrix-Vektor Multiplikation, explizit oder implizit (trivial: jeder Eintrag im Ergebnisvektor kann unabhängig berechnet werden)
- Skalarprodukt: parallele Reduktion
- Normen-Berechnung: parallele Reduktion

Gewöhnliche Differentialgleichungen:

- … sind Differentialgleichungen, bei denen nur Ableitungen nach einer Variablen vorkommen
- Ordnung: höchste vorkommende Ableitung
- Anfangswertproblem: zu gegebenem x muss Bedingung erfüllt sein, z.B. $y(x_0) = y_0, y'(x_0) = y_1, \dots$
- Gewöhnliche Differentialgleichungen beliebiger Ordnung n lassen sich auf ein System von n gewöhnlichen Differentialgleichungen erster Ordnung umformen
- Löser für $\dot{y} = f(t, y) \;\; , \;\; y(t_0) = y_0$:
    - Einschritt-Verfahren:
        - Expliziter Euler: $y_{k+1} = y_k + hf(t_k, y_k) \;\; , \;\; t_k = t_0 + kh$
        - Implizierter Euler: $y_{k+1} = y_k + hf(t_{k+1}, y_{k+1}) \;\; , \;\; t_k = t_0 + kh$
        - Klassische Runge-Kutta-Methode ($f(t, y)$ an mehreren Stellen auswerten um auf den nächsten Wert zu kommen):
          $$y_{k+1} = y_k + \frac{1}{6}g_1 + \frac{1}{3}g_2 + \frac{1}{3}g_3 + \frac{1}{6}g_4 \quad \text{mit} \quad g_{1-4} = hf(\#, \#)$$
        - Konsistenz- und Konvergenzordnung:
            - Expliziter und impliziter Euler: O(h)
            - Klassische Runge-Kutta-Methode: $O(h^4)$
    - Mehrschritt-Verfahren:
        - Zum nächsten Schritt mehr als nur den vorherigen Schritt mit einbeziehen → höhere Konsistenzordnungen
        - Mittelpunkt-Methode:
            - $y_{k+1} = y_{k-1} + 2hf(t_k, y_k)$
            - Explizite Zwei-Schritt-Methode der Ordnung 2
        - Milne:
            - $y_{k+1} = y_{k-3} + \frac{4h}{3}(2f(t_k, y_k) - f(t_{k-1}, y_{k-1}) + 2f(t_{k-2}, y_{k-2}))$
            - Explizite Vier-Schritt-Methode der Ordnung 4
        - Adams-Bashforth:
            - $y_{k+1} = y_k + \frac{h}{24}(55f(t_k, y_k) - 59f(t_{k-1}, y_{k-1}) + 37f(t_{k-2}, y_{k-2}) - 9f(t_{k-3}, y_{k-3}))$
            - Explizite Vier-Schritt-Methode der Ordnung 4
        - Simpson:
            - $y_{k+1} = y_{k-1} + \frac{h}{3}(f(t_{k+1}, y_{k+1}) + 4f(t_k, y_k) + f(t_{k-1}, y_{k-1}))$
            - Implizite Zwei-Schritt-Methode der Ordnung 4