Xeon Phi: Feeding the Monster

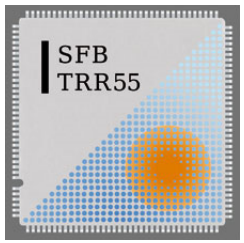
How to deal with Many Cores

Florian Rappl

Institute for Theoretical Physics

23.04.2015

Universität Regensburg



Studienstiftung
des deutschen Volkes

Studienstiftung
des deutschen Volkes

Was?

Wer?

- Doktorand theoretische Teilchenphysik
- Spezialisiert auf Web, LoB und HPC Anwendungen
- Algorithmen für Quantenchromodynamik (QCD)
- High Performance Computing (v.a. Gitter QCD)
- Spezialisierte Supercomputer (QPACE, iDataCool)
- Aktuell: QPACE 2

Outline

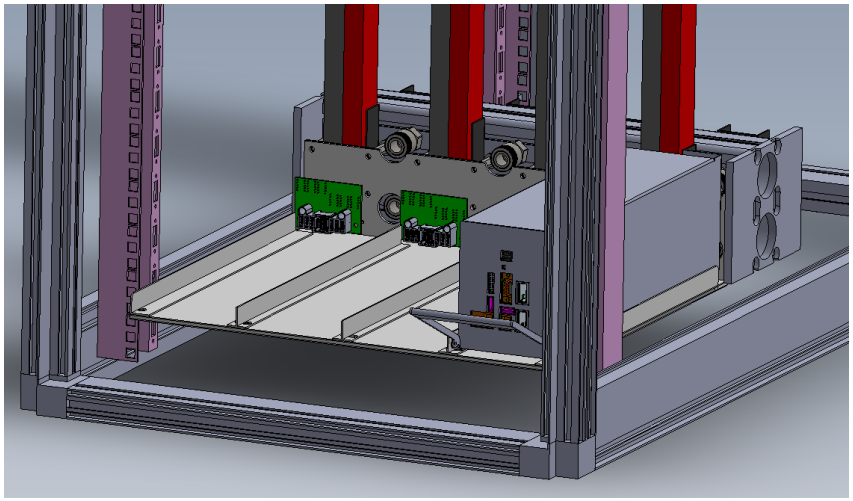
Outline

Outline

QPACE 2 Projekt

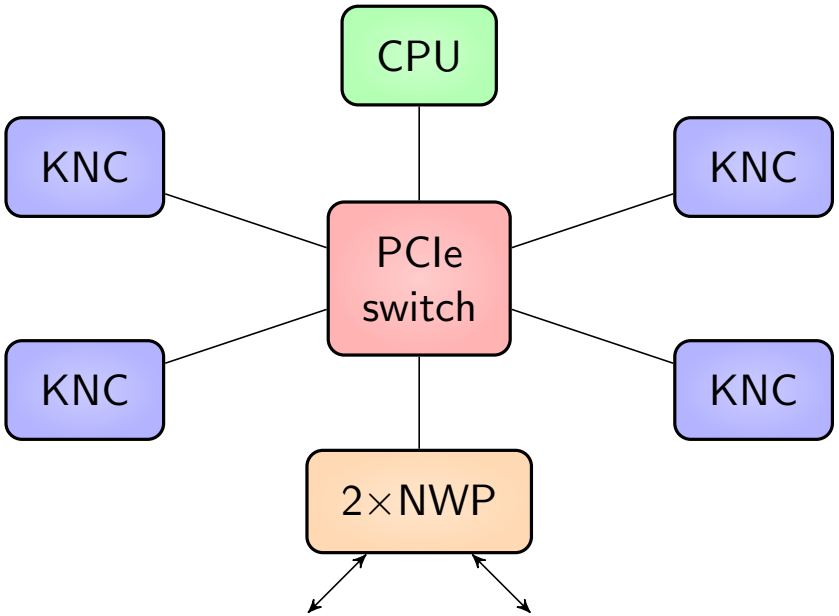
- Nachfolge von QPACE (#1 Green-500)
- Modulares Design (z.B. Phi $\leftrightarrow$ GPU)
- Warmwasserkühlung (iDataCool, SuperMUC)
- Simulation von Quantenchromodynamik
- Verwendung von Standardkomponenten
- Performance $\sim$300 TFlop/s (Peak)
- Maximale Leistungsaufnahme $\sim$75 kW

Das Rack



- Stromverteilung (PDU, PSU)
- Infiniband Switches
- Netzwerktechnik
- Wasserverteilung
- 64 Einheiten (Bricks) in 24U
- Gesamte Höhe des Racks 42U

Ein Brick

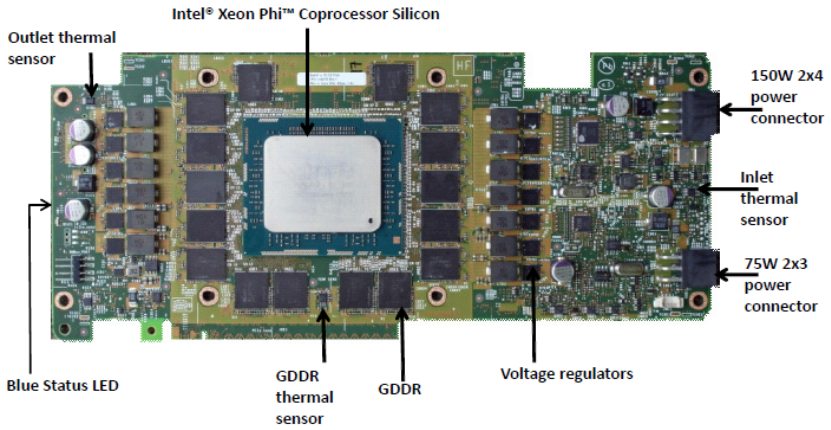


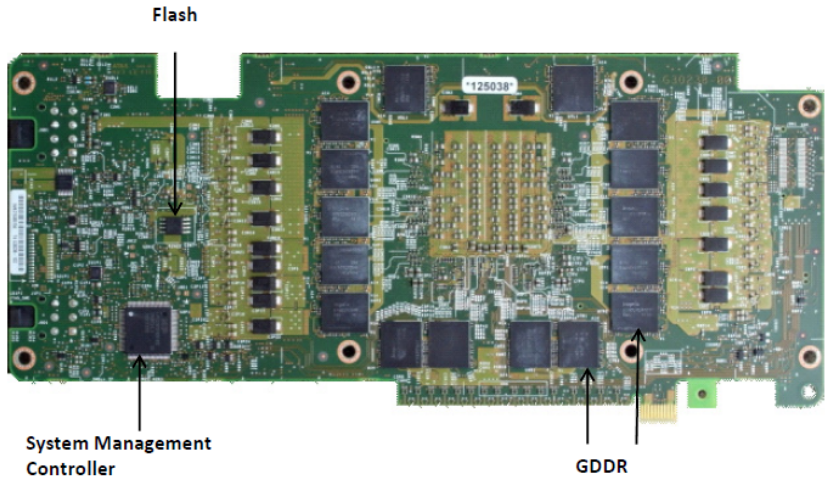
- Eine ConnectIB Karte (Dualport)
- Motherboard mit FDR PCIe Switch
- CPU-Karte (Intel E3, 16 GB)
- 4 Recheneinheiten (Intel Xeon Phi)
- Wasseranschluss und Verteilung

Ein Brick

Eine Recheneinheit

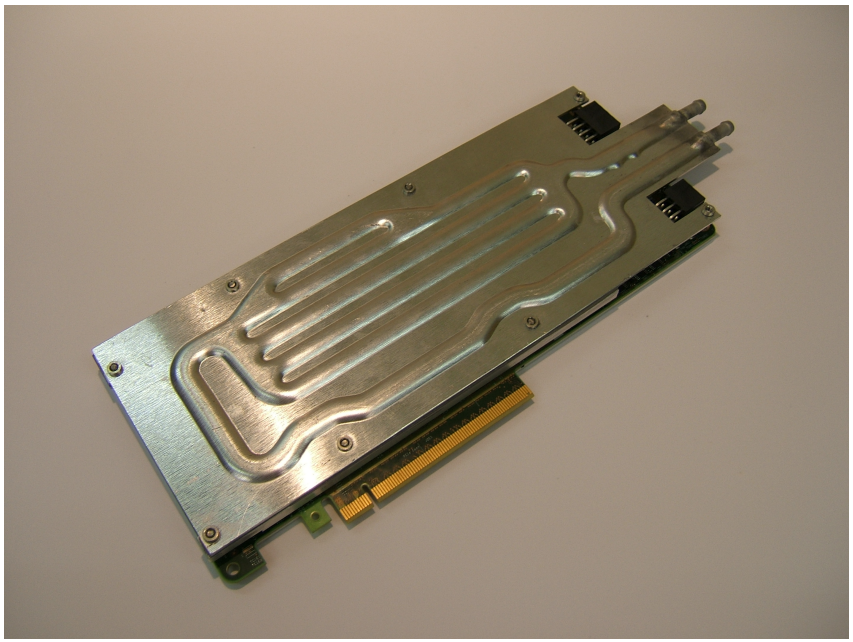
- Intel Xeon Phi 7120X
- 61 Kerne
- 4 HW Threads
- 16 GB GDDR RAM
- 1.238 GHz
- Bandbreite ca. 170 GB/s





Outline

Wasserkühlung



- Transport über Rollbonds
- Verbindung über Interposer
- Essentiell für geringe Energiekosten

Outline

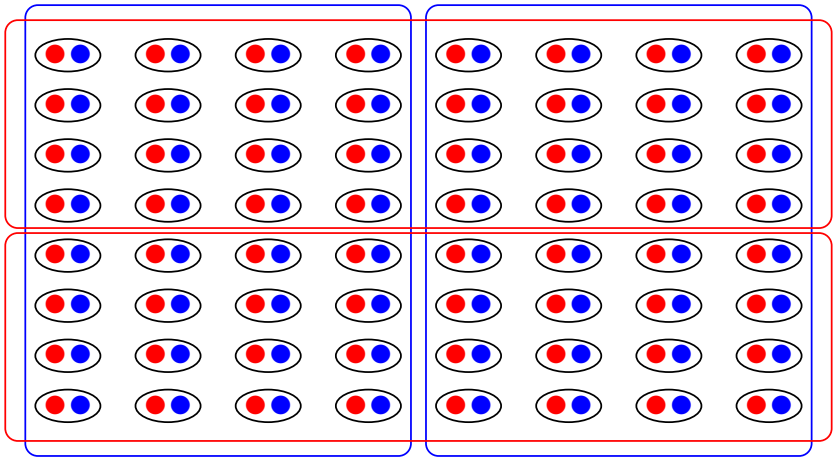
Kommunikationsebenen

- Es existieren zwei Kommunikationsebenen
  1. Intra-Brick
  2. Inter-Brick
- Intra-Brick über PCIe
- Inter-Brick baut auf InfiniBand auf
- Differenzierung (im Code) notwendig

Kommunikationsebenen

- Es existieren zwei Kommunikationsebenen
  1. Intra-Brick
  2. Inter-Brick
- Intra-Brick über PCIe
- Inter-Brick baut auf InfiniBand auf
- Differenzierung (im Code) notwendig

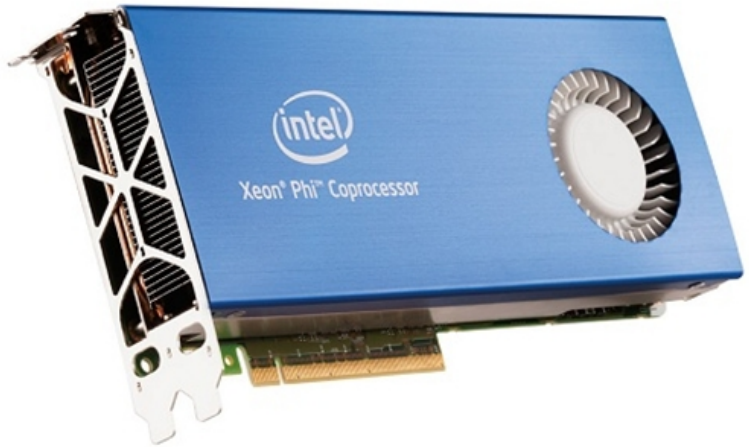
Netzwerk-Topologie



- Hyper-Crossbar Topologie
- Kommunikation der 256 KNCs
- 4 KNCs teilen sich 2 Ports
- 4 Switches für 128 Kabel
- Max. Hopping 2 Switches

Outline

Many Integrated Cores



- Bessere Leistung pro Watt
- Verwendet wie eine GPU, aber ähnlich zu CPU
- Erste verfügbare Version: Intel Xeon Phi (*Knights Corner*)
- $\sim$1 TFlop/s (theoretisch, double prec.)
- Zukunft: Knights Landing, Knights Hill

Outline

Überblick

- In-order Ausführung
- `float32` und `float64` Anweisungen
- Zentrale Memory Controller in Ringbus
- Zwei Threads wechseln sich immer ab
- Effektive Frequenz daher ∼500 MHz
- Div. Tricks wie FMA (*Fused Multiply and Add*)
- Hyper-Threading
- 1 Core für OS (Linux)
- Kommunikation über SCIF

Überblick

- In-order Ausführung
- `float32` und `float64` Anweisungen
- Zentrale Memory Controller in Ringbus
- Zwei Threads wechseln sich immer ab
- Effektive Frequenz daher ∼500 MHz
- Div. Tricks wie FMA (*Fused Multiply and Add*)
- Hyper-Threading
- 1 Core für OS (Linux)
- Kommunikation über SCIF

Überblick

- In-order Ausführung
- `float32` und `float64` Anweisungen
- Zentrale Memory Controller in Ringbus
- Zwei Threads wechseln sich immer ab
- Effektive Frequenz daher ∼500 MHz
- Div. Tricks wie FMA (*Fused Multiply and Add*)
- Hyper-Threading
- 1 Core für OS (Linux)
- Kommunikation über SCIF

Speicherverwaltung

- Kleiner (32 kB) gemeinsamer L1 Cache
- Geringer Instruction Cache
- L2 Cache mit 512 kB
- MESI Protokoll für Cache-Kohärenz
- Global verteiltes Tag Directory
- Kein L1 Prefetcher
- Mittelmäßiger L2 Prefetcher

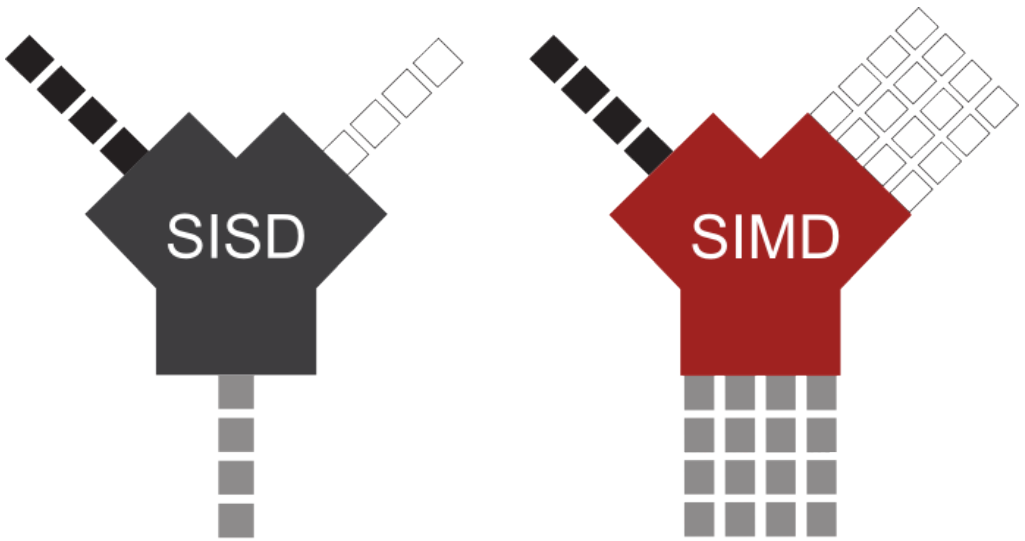
Speicherverwaltung

- Kleiner (32 kB) gemeinsamer L1 Cache
- Geringer Instruction Cache
- L2 Cache mit 512 kB
- MESI Protokoll für Cache-Kohärenz
- Global verteiltes Tag Directory
- Kein L1 Prefetcher
- Mittelmäßiger L2 Prefetcher

Vektoreinheit

- z.B. SSE (128-bit Vektor), AVX (256-bit Vektor)
- KNC: 16 `float32` (512-bit) breit
- Separate Register (getrennt von der Skalareinheit)
- Spezielle Instruktionen (kein SSE, MMX, AVX, ...)
- Umwandlungen (float32 $\leftrightarrow$ `float64`) vorhanden
- Spezielle Operationen ($\sin$, $\cos$, ...) auch für die VPU

Outline



- Eine Anweisung, viele In- und Outputs
- Alle Loads zuerst, alle Stores zuletzt
- Compiler Intrinsics (z.B. `_mm512_fmadd_ps`)
- Auto-Vektorisierung (ab `-O2`)
- Nicht möglich mit Anweisungssprüngen
- Analysen aktivieren mit `-qopt-report-phase=vec`

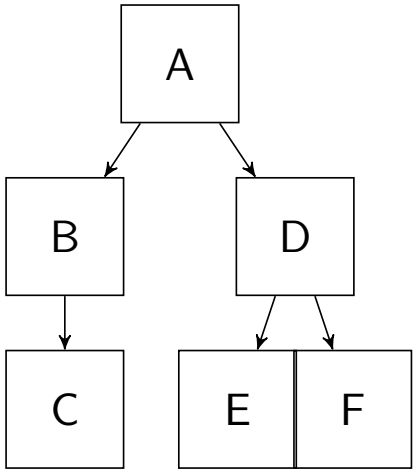
Data-Oriented Design

- Datentransformationen im Fokus
- Gruppieren nach Verwendung
- Struktur und Häufigkeit beachten
- Bessere Performance durch Alignment
- Maximierung der Wiederverwendung von Cachelines
- Cacheline Optimierungen essentiell

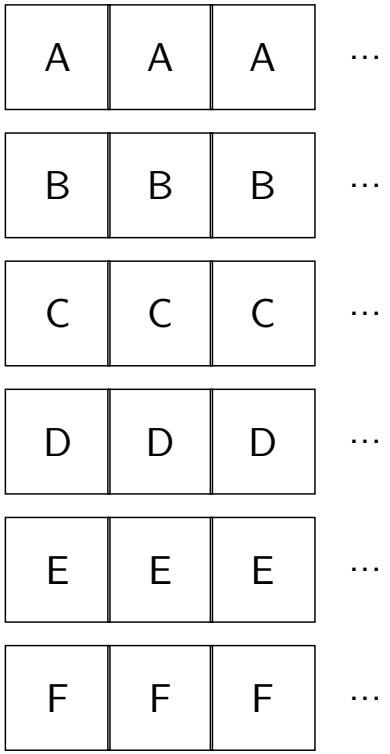
Data-Oriented Design

- Datentransformationen im Fokus
- Gruppieren nach Verwendung
- Struktur und Häufigkeit beachten
- Bessere Performance durch Alignment
- Maximierung der Wiederverwendung von Cachelines
- Cacheline Optimierungen essentiell

Object-Oriented vs Data-Oriented

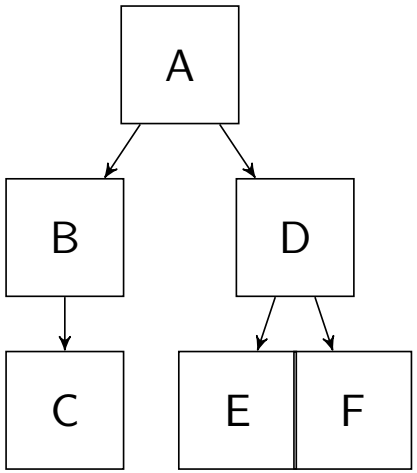


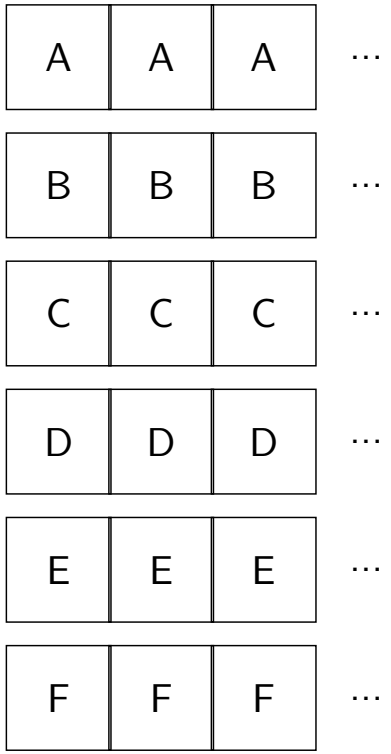
Potentiell viele Cache Misses
(A, B, C, . . . , F, A, B, . . .)



Optimierte Aufrufsreihenfolge
(A, A, A, . . . , B, B, B, . . .)

Object-Oriented vs Data-Oriented

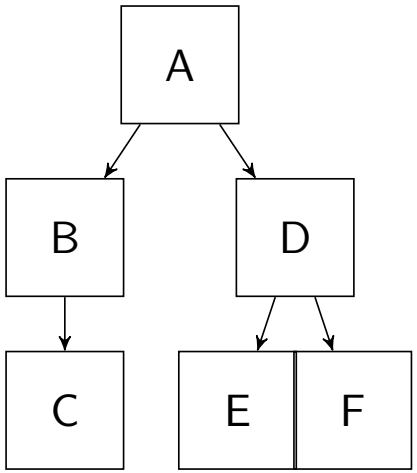




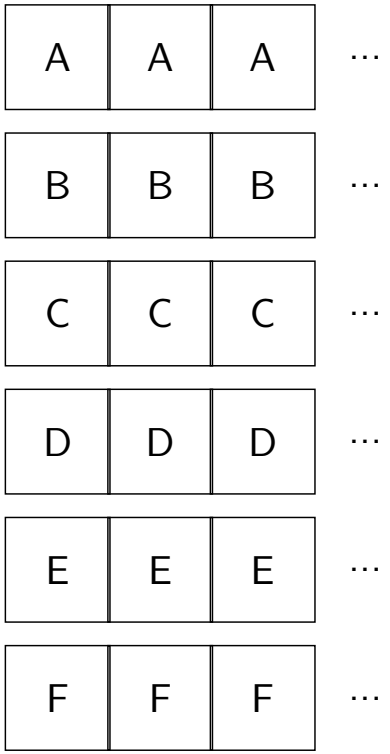
Potentiell viele Cache Misses
(A, B, C, . . . , F, A, B, . . .)

Optimierte Aufrufsreihenfolge
(A, A, A, . . . , B, B, B, . . .)

Object-Oriented vs Data-Oriented



Potentiell viele Cache Misses
(A, B, C, . . . , F, A, B, . . .)



Optimierte Aufrufsreihenfolge
(A, A, A, . . . , B, B, B, . . .)

Outline

Scaling Architecture

- Optimale Ausnutzung vorhandener Resourcen
- Erhaltung der Portabilität
- Vektorisierung falls möglich
- Parallelisierung falls gewünscht
- Skalierende Verteilung (Distribution) möglich

$\Rightarrow$ Wir brauchen eine effiziente Architektur

Scaling Architecture

- Optimale Ausnutzung vorhandener Resourcen
- Erhaltung der Portabilität
- Vektorisierung falls möglich
- Parallelisierung falls gewünscht
- Skalierende Verteilung (Distribution) möglich

$\Rightarrow$ Wir brauchen eine effiziente Architektur

Outline

Möglichkeiten

- Parallelisierung
  - Intel Cilk Plus
  - OpenMP
  - MPI (interessant für Distribution)
  - Intel Threading Building Blocks
  - PThreads
  - *u.v.m.*
- Vektorisierung
  - Automatisch
  - Compiler Pragmas
  - Intrinsics / Assembler
  - Intel Cilk Plus
  - OpenMP
  - *u.v.m.*

Unsere Wahl

- Entwickelt von Matthias Kretz (Universität Frankfurt) [1]
- Framework für explizite Vektorisierung
- Aktuelle Version 0.7.4
- Verschiedene Architekturen vorhanden (AVX, SSE, Phi)
- Funktioniert mit allen gängigen (C++11) Compilern
- Fallback: Skalare Operationen
- Datentypen sind STL kompatibel

- Seit einem Jahr keine Weiterentwicklung mehr

[1] `http://code.compeng.uni-frankfurt.de/projects/vc/`

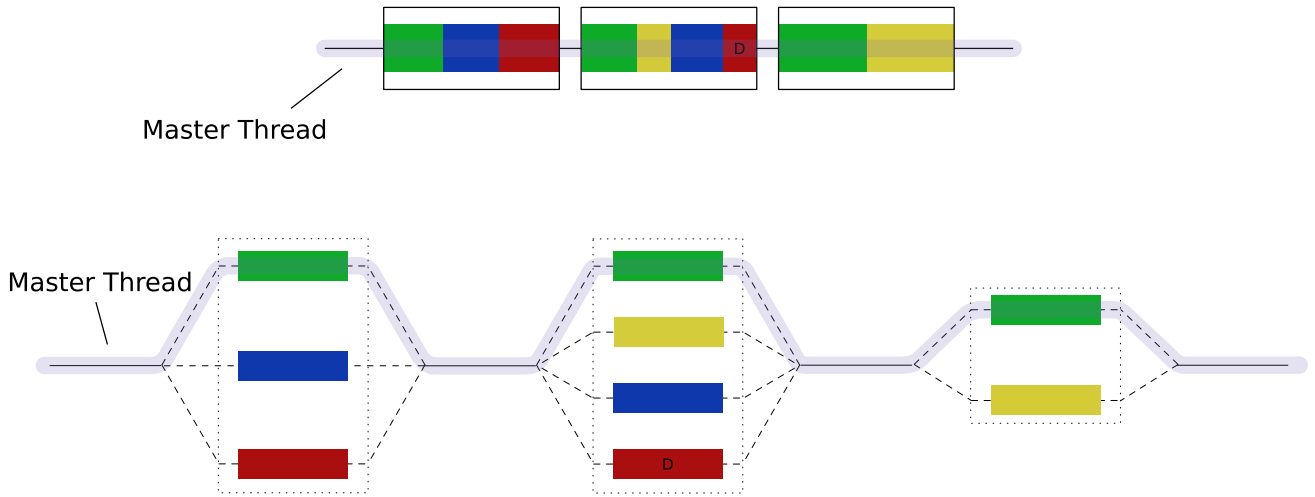
- Entwickelt von Matthias Kretz (Universität Frankfurt) [1]
- Framework für explizite Vektorisierung
- Aktuelle Version 0.7.4
- Verschiedene Architekturen vorhanden (AVX, SSE, Phi)
- Funktioniert mit allen gängigen (C++11) Compilern
- Fallback: Skalare Operationen
- Datentypen sind STL kompatibel

- Seit einem Jahr keine Weiterentwicklung mehr

[1] `http://code.compeng.uni-frankfurt.de/projects/vc/`

Vc: Beispiel

```
template<typename T, unsigned int Size>
class Matrix {
  typedef Vc::Vector<T> V;
  public:
    Matrix& operator=(const T& val) {
      V vec(val);
      for (auto i = 0u; i < m_mem.vectorsCount(); ++i)
        m_mem.vector(i) = vec;
      return *this;
    }
    Matrix& operator+=(const Matrix& rhs) {
      for (auto i = 0u; i < m_mem.vectorsCount(); ++i) {
        V v1(m_mem.vector(i));
        v1 += V(rhs.m_mem.vector(i));
        m_mem.vector(i) = v1;
      }
      return *this;
    }
  private:
    Vc::Memory<V, Size * Size> m_mem;
};
```



- Verwendet Fork-Join Modell
- Elegante und einfache Parallelisierung
- Trivial für Schleifen
- OpenMP 4.0 führt SIMD ein

OpenMP: Beispiel

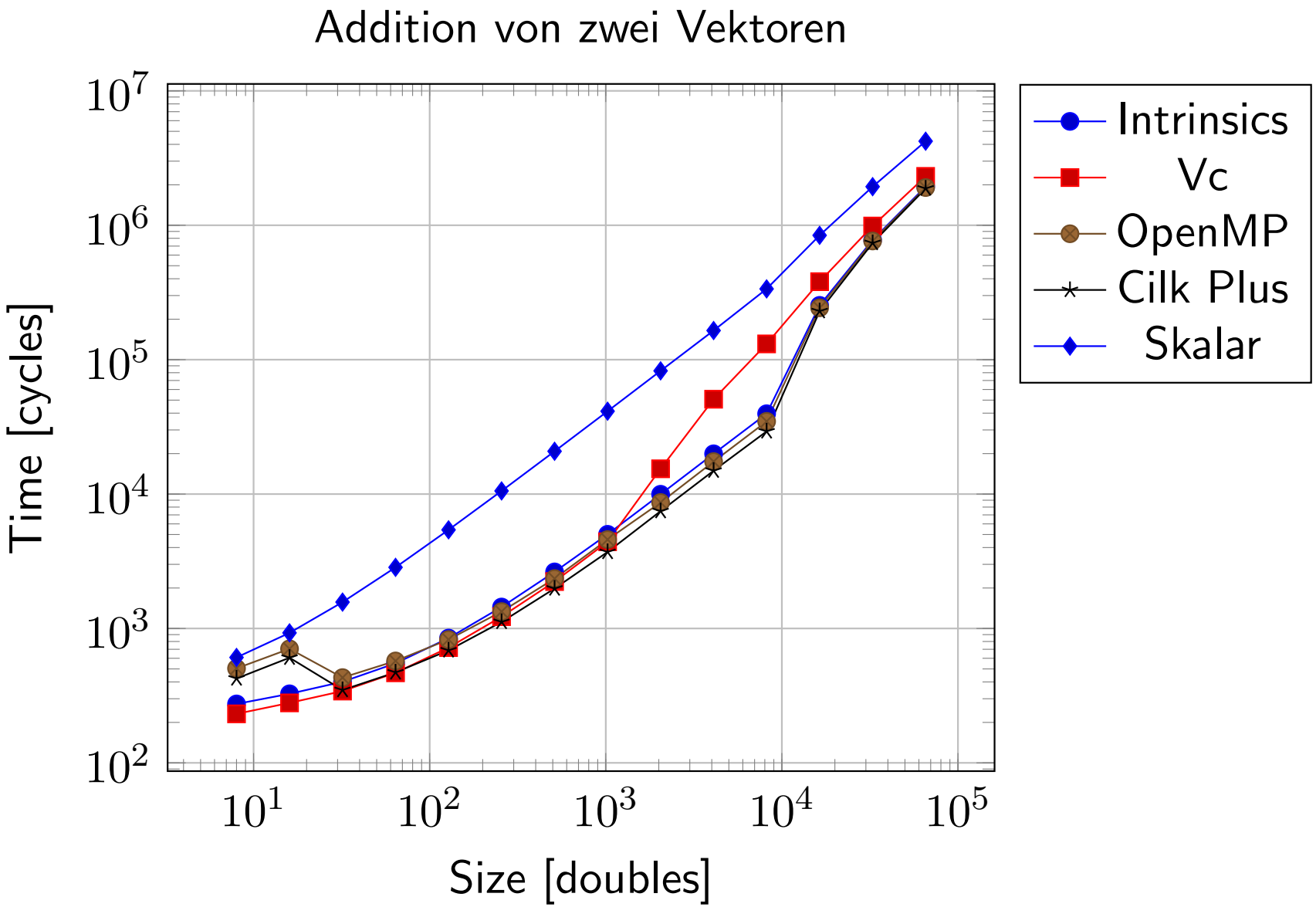
```
template<typename T, unsigned int Size>
class Matrix {
  public:
    Matrix& operator=(const T& val) {
      #pragma omp simd
      for (auto i = 0u; i < Size * Size; ++i)
        m_mem[i] = val;
      return *this;
    }
    Matrix& operator+=(const Matrix& rhs) {
      #pragma omp simd
      for (auto i = 0u; i < Size * Size; ++i)
        m_mem[i] += rhs.m_mem[i];
      return *this;
    }
  private:
    T m_mem[Size * Size];
};
```

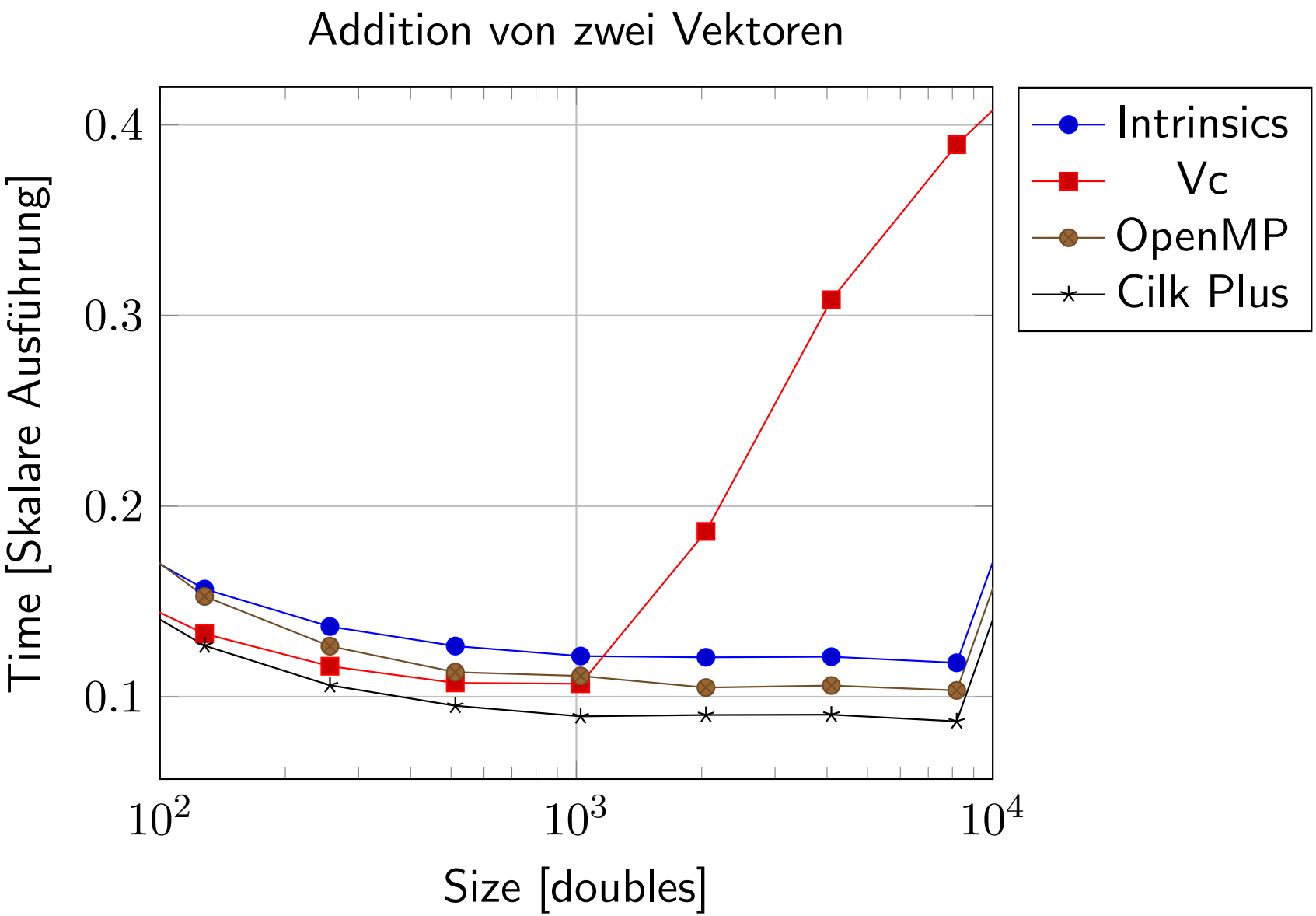
Cilk Plus

- Spracherweiterung von C++
- Ebenfalls Fork-Join Modell
- Drei (sinnvolle) Schlüsselwörter:
  - `spawn`
  - `sync`
  - `cilk_for`
- SIMD durch erweiterte Array Notation
- Implementiert in GCC (ab 4.9), ICC, Clang (Fork)

Cilk Plus: Beispiel

```
template<typename T, unsigned int Size>
class Matrix {
  public:
    Matrix& operator=(const T& val) {
      m_mem[0:Size * Size] = val;
      return *this;
    }
    Matrix& operator+=(const Matrix& rhs) {
      m_mem[0:Size * Size] += rhs.m_mem[0:Size * Size];
      return *this;
    }
  private:
    T m_mem[Size * Size];
};
```


Addition von zwei Vektoren
Time [cycles]
Size [doubles]
10^2
10^3
10^4
10^5
10^6
10^7
10^1
10^2
10^3
10^4
10^5
Intrinsics
Vc
OpenMP
Cilk Plus
Skalar


Addition von zwei Vektoren
Time [Skalare Ausführung]
0.4
0.3
0.2
0.1
10^2
10^3
10^4
Size [doubles]
Intrinsics
Vc
OpenMP
Cilk Plus

Outline

Definition

Computation Invocation Is Distribution

Analog zu RAII

- Geringst möglicher Overhead
- Gute Kontrolle
- Hohe Portabilität

⇝ Regelt Vektorisierung, Parallelisierung und Distribution

! Alles optional und steuerbar

Definition
Computation Invocation Is Distribution

Analog zu RAII

- Geringst möglicher Overhead
- Gute Kontrolle
- Hohe Portabilität

⇝ Regelt Vektorisierung, Parallelisierung und Distribution

! Alles optional und steuerbar

Definition
Computation Invocation Is Distribution

Analog zu RAII

- ➤ Geringst möglicher Overhead
- ➤ Gute Kontrolle
- ➤ Hohe Portabilität

⇝ Regelt Vektorisierung, Parallelisierung und Distribution

! Alles optional und steuerbar

Definition
Computation Invocation Is Distribution

Analog zu RAII

- ➤ Geringst möglicher Overhead
- ➤ Gute Kontrolle
- ➤ Hohe Portabilität

$\rightsquigarrow$ Regelt Vektorisierung, Parallelisierung und Distribution

! Alles optional und steuerbar

Templates

- Ziel: Automatische Erzeugung von Code
- Vollständig transparente Abstraktion
- Meta-Programmierung essentiell
- C++ im Vorteil: Verwendung von TMP
- Einsatz von Pseudo-Klassen und Traits
- Transport von Datentypen (SIMD)
- Regelung von Parallelisierung und Distribution
- Entkopplung von konkreten Technologien

Templates

- Ziel: Automatische Erzeugung von Code
- Vollständig transparente Abstraktion
- Meta-Programmierung essentiell
- C++ im Vorteil: Verwendung von TMP
- Einsatz von Pseudo-Klassen und Traits
- Transport von Datentypen (SIMD)
- Regelung von Paralleliserung und Distribution
- Entkopplung von konkreten Technologien

Templates

- Ziel: Automatische Erzeugung von Code
- Vollständig transparente Abstraktion
- Meta-Programmierung essentiell
- C++ im Vorteil: Verwendung von TMP
- Einsatz von Pseudo-Klassen und Traits
- Transport von Datentypen (SIMD)
- Regelung von Parallelisierung und Distribution
- Entkopplung von konkreten Technologien

Intrinsics

- Bevorzugt gg. Assembler, OpenMP, ...
- Möglichkeit SIMD voll auszunutzen
- Sollten wg. Portabilität abstrahiert werden
- Folgerichtig eigener Allocator notwendig
- Elementare Funktionen (z.B. $\sin$)?
- Elementare Operationen (z.B. $+$)?

$\Rightarrow$ Orientierung an Umsetzung von Vc

Intrinsics

- Bevorzugt gg. Assembler, OpenMP, ...
- Möglichkeit SIMD voll auszunutzen
- Sollten wg. Portabilität abstrahiert werden
- Folgerichtig eigener Allocator notwendig
- Elementare Funktionen (z.B. $\sin$)?
- Elementare Operationen (z.B. $+$)?

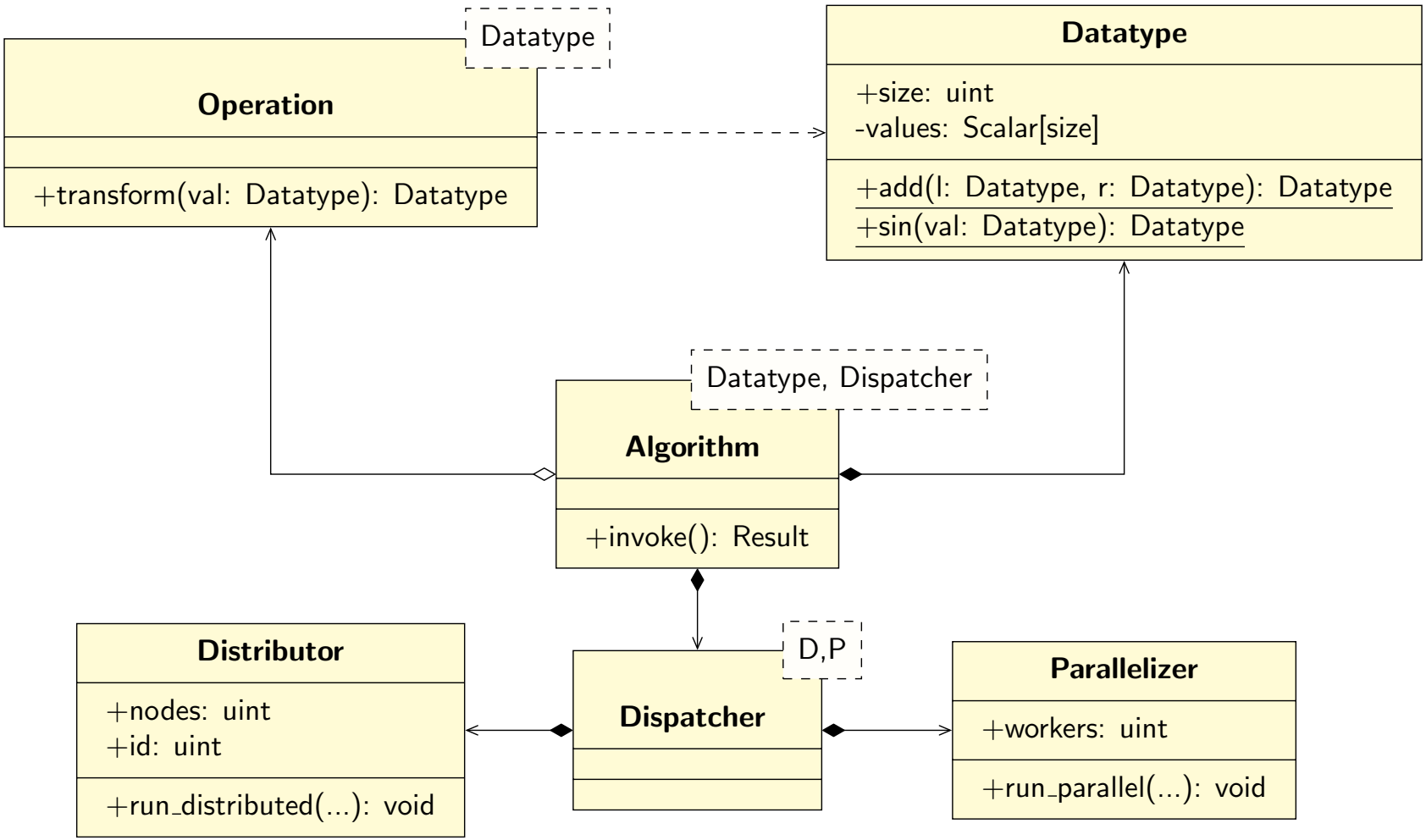
$\Rightarrow$ Orientierung an Umsetzung von Vc

Intrinsics

- Bevorzugt gg. Assembler, OpenMP, ...
- Möglichkeit SIMD voll auszunutzen
- Sollten wg. Portabilität abstrahiert werden
- Folgerichtig eigener Allocator notwendig
- Elementare Funktionen (z.B. $\sin$)?
- Elementare Operationen (z.B. $+$)?

$\Rightarrow$ Orientierung an Umsetzung von Vc

Schematischer Aufbau



Outline

Lösung eines Gleichungssystems

- Parallelisierung einer GMRES Implementierung
- Intrinsics für Vektorisierung
- OpenMP für Parallelisierung
- MPI für Distribution
- Effiziente Aufteilung notwendig
  - Distributed:
    Höchste Laufzeit mit geringster Kommunikation
  - Parallel:
    Mittlere Laufzeit mit häufigerer Kommunikation
  - SIMD:
    Geringe Laufzeit mit permanenter Kommunikation

Lösung eines Gleichungssystems

- Parallelisierung einer GMRES Implementierung
- Intrinsics für Vektorisierung
- OpenMP für Parallelisierung
- MPI für Distribution
- Effiziente Aufteilung notwendig
  - Distributed:
    Höchste Laufzeit mit geringster Kommunikation
  - Parallel:
    Mittlere Laufzeit mit häufigerer Kommunikation
  - SIMD:
    Geringe Laufzeit mit permanenter Kommunikation

Lösung eines Gleichungssystems

- Parallelisierung einer GMRES Implementierung
- Intrinsics für Vektorisierung
- OpenMP für Parallelisierung
- MPI für Distribution
- Effiziente Aufteilung notwendig
  - Distributed:
    Höchste Laufzeit mit geringster Kommunikation
  - Parallel:
    Mittlere Laufzeit mit häufigerer Kommunikation
  - SIMD:
    Geringe Laufzeit mit permanenter Kommunikation

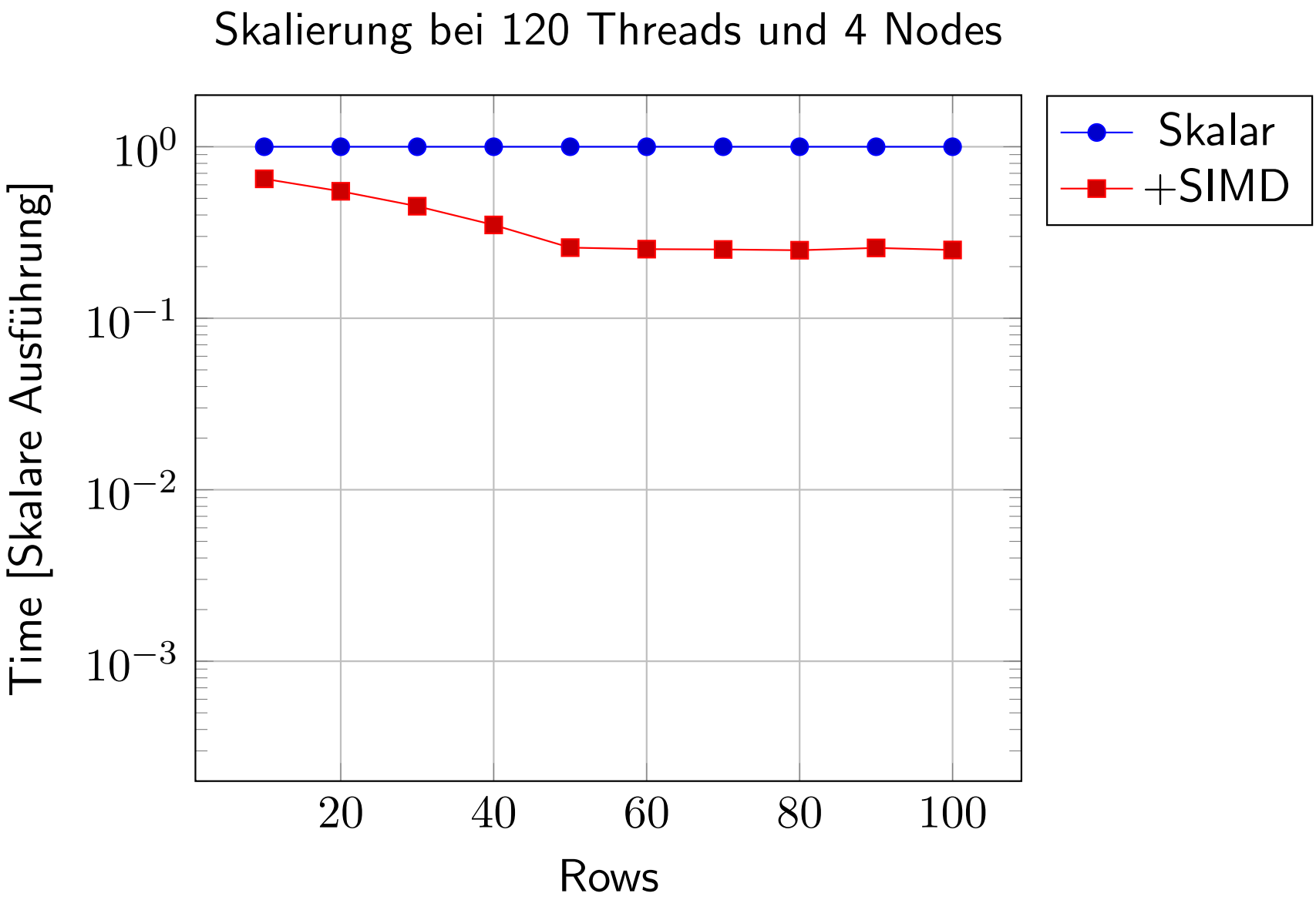
Lösung eines Gleichungssystems

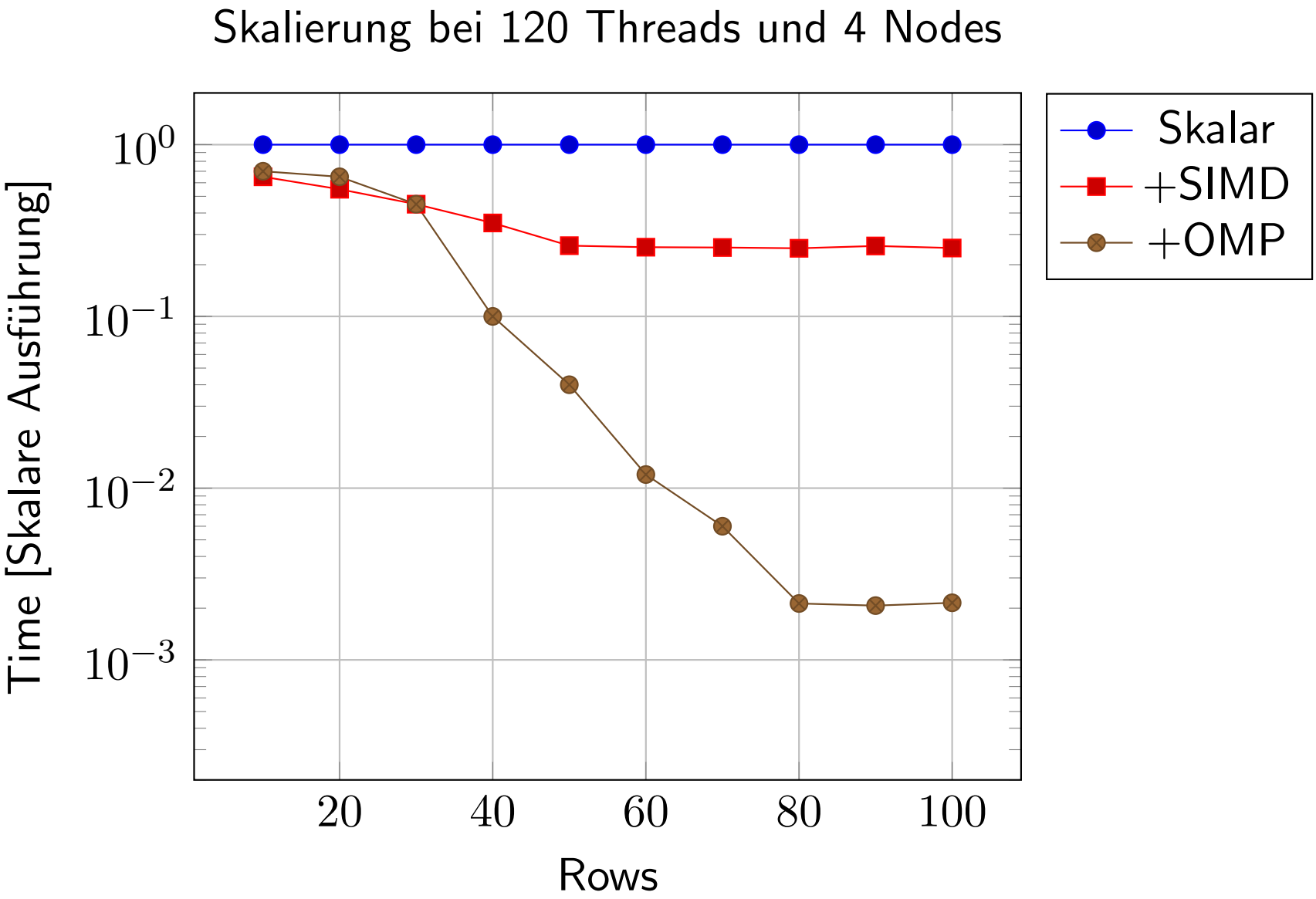
- Parallelisierung einer GMRES Implementierung
- Intrinsics für Vektorisierung
- OpenMP für Parallelisierung
- MPI für Distribution
- Effiziente Aufteilung notwendig
  - Distributed:
    Höchste Laufzeit mit geringster Kommunikation
  - Parallel:
    Mittlere Laufzeit mit häufigerer Kommunikation
  - SIMD:
    Geringe Laufzeit mit permanenter Kommunikation

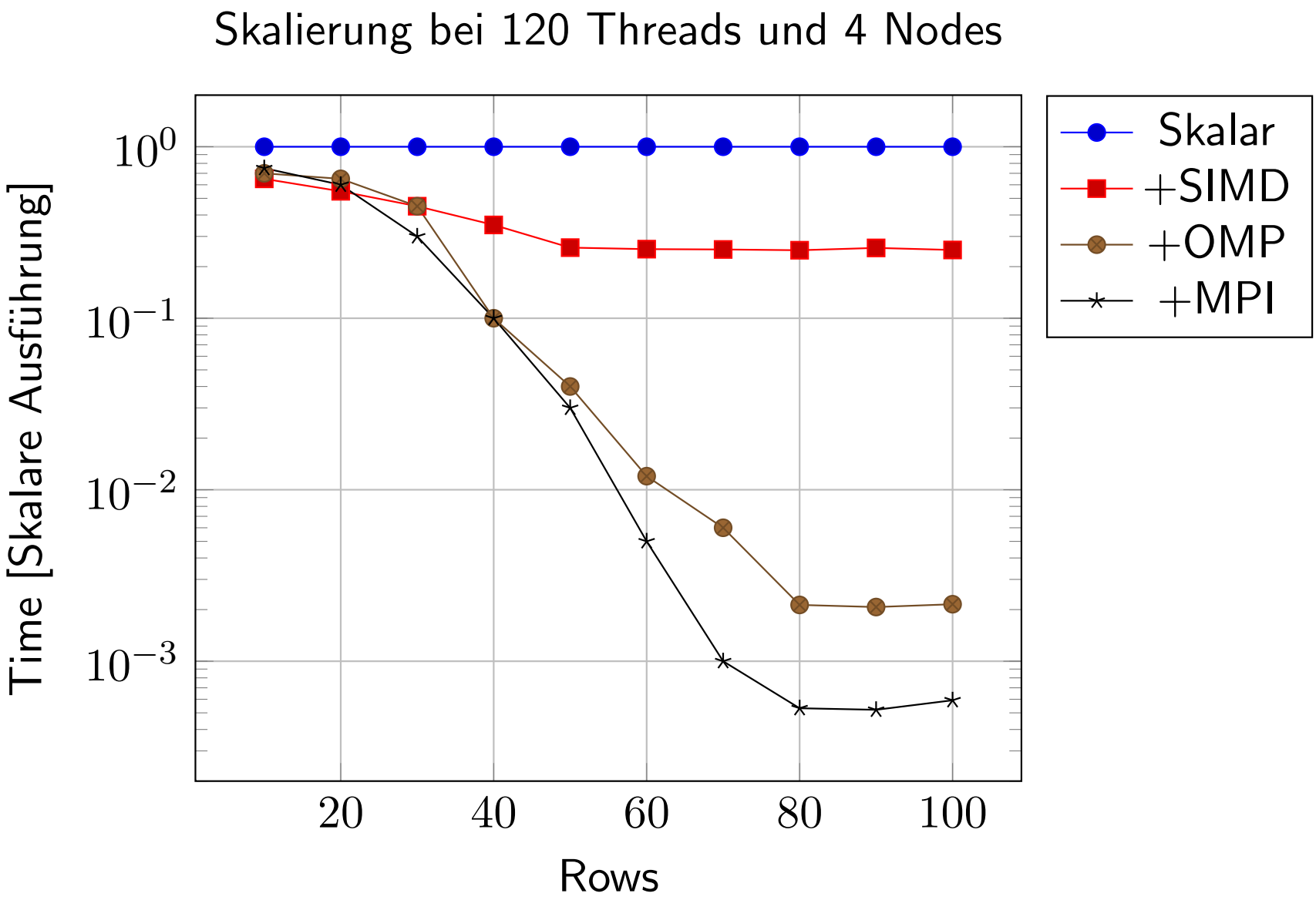
Lösung eines Gleichungssystems

- Parallelisierung einer GMRES Implementierung
- Intrinsics für Vektorisierung
- OpenMP für Parallelisierung
- MPI für Distribution
- Effiziente Aufteilung notwendig
  - Distributed:
    Höchste Laufzeit mit geringster Kommunikation
  - Parallel:
    Mittlere Laufzeit mit häufigerer Kommunikation
  - SIMD:
    Geringe Laufzeit mit permanenter Kommunikation

Benchmarks


Skalierung bei 120 Threads und 4 Nodes
Time [Skalare Ausführung]
10^0
10^{-1}
10^{-2}
10^{-3}
20
40
60
80
100
Rows
Skalar
+SIMD


Skalierung bei 120 Threads und 4 Nodes
Time [Skalare Ausführung]
10^0
10^{-1}
10^{-2}
10^{-3}
20
40
60
80
100
Rows
Skalar
+SIMD
+OMP


Skalierung bei 120 Threads und 4 Nodes
Time [Skalare Ausführung]
10^0
10^{-1}
10^{-2}
10^{-3}
20
40
60
80
100
Rows
Skalar
+SIMD
+OMP
+MPI

Outline

Zusammenfassung

- QPACE 2 guter Test für Skalierbarkeit
- Vektorunit essentiell für Performance
- Richtiges Datenlayout entscheidet
- Datenstrukturen sind wichtig
- Abstraktion ist notwendig, muss transparent sein
- Entkopplung der Parallelisierungsstrategie

Zusammenfassung

- QPACE 2 guter Test für Skalierbarkeit
- Vektorunit essentiell für Performance
- Richtiges Datenlayout entscheidet
- Datenstrukturen sind wichtig
- Abstraktion ist notwendig, muss transparent sein
- Entkopplung der Parallelisierungsstrategie

Zusammenfassung

- QPACE 2 guter Test für Skalierbarkeit
- Vektorunit essentiell für Performance
- Richtiges Datenlayout entscheidet
- Datenstrukturen sind wichtig
- Abstraktion ist notwendig, muss transparent sein
- Entkopplung der Parallelisierungsstrategie

Vielen Dank!

Mitwirkende am QPACE 2 Projekt

- Regensburg: Jacques Bloch, Johann Deinhart, Benjamin Glässle, Simon Heybrock, Robert Lohmayer, Simon Mages, Bernhard Mendl, Nils Meyer, Stefan Solbrig, Norbert Sommer, Tilo Wettig
- Jülich: Dirk Pleiter
- Eurotech: Paul Arts, Alessio Parcianello, Mauro Rossi, Giampietro Techiolli, Gianpaolo Zanier
- Advanet: Yu Komatsubara