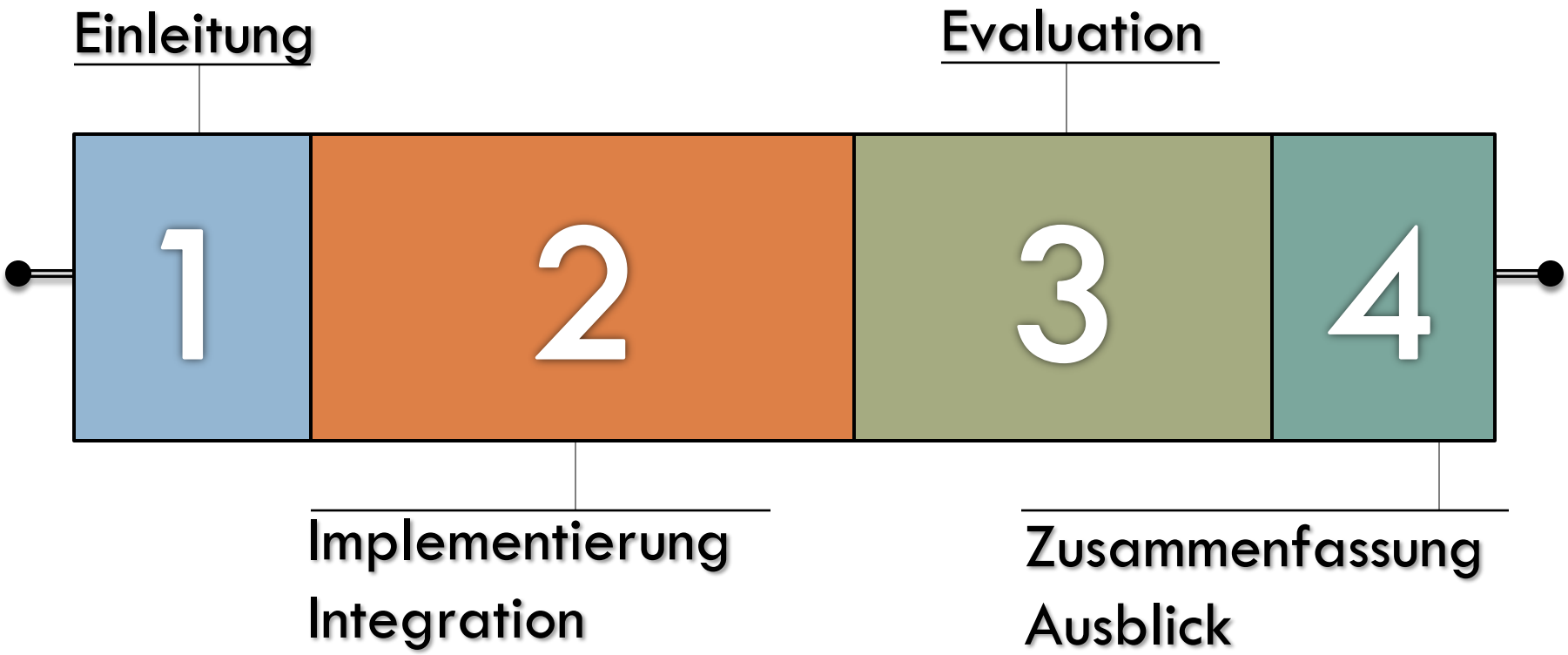
Exploiting Object-Orientation to Parallelize and Optimize C++ Applications

Christopher Schleiden – Bachelor Kolloquium – 15.09.2009


Einleitung
Evaluation
1
2
3
4
Implementierung
Integration
Zusammenfassung
Ausblick

Einleitung

laperf

- Lineare Algebra Bibliothek für C++
- Möglichkeit zur Integration in C++ Programme
  - Nutzung Objekt-Orientierter Prinzipien
  - Generische Programmierung: Templates
- Unterstützung verschiedener Parallelisierungsparadigmen
  - OpenMP
  - Intel Threading Building Blocks
  - ...

laperf

- Verschiedene Datentypen
  - Dichte Matrizen
  - Dünnbesetzte Matrizen
  - Vektoren
- Operationen auf Datentypen
  - Matrix-Vektor Multiplikation
  - Skalarprodukt zweier Vektoren
  - Addition/Subtraktion/... von Vektoren
  - ...

laperf – Verwendung

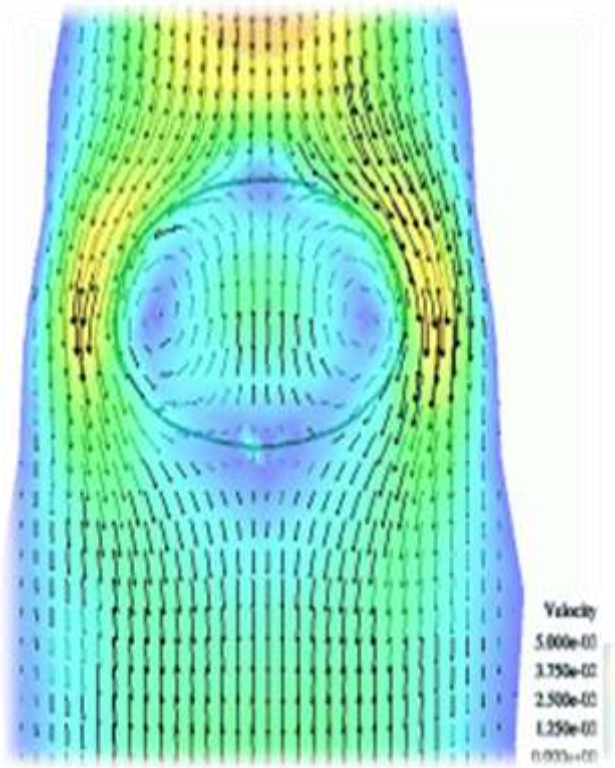
- Angelehnt an mathematische Notation

```
laperf::vector<double, OMPInternalPar> x,b,r;
laperf::matrix_crs<double> A;
```

```
r = b - A * x;
double res = r.norm_2();
```

DROPS

- Adaptiver Navier-Stokes Löser
- Simulation mehrphasiger reaktiver Strömungen
- Modellierung:
  Finite Elemente Methode
- Generischer, komplexer C++ Code

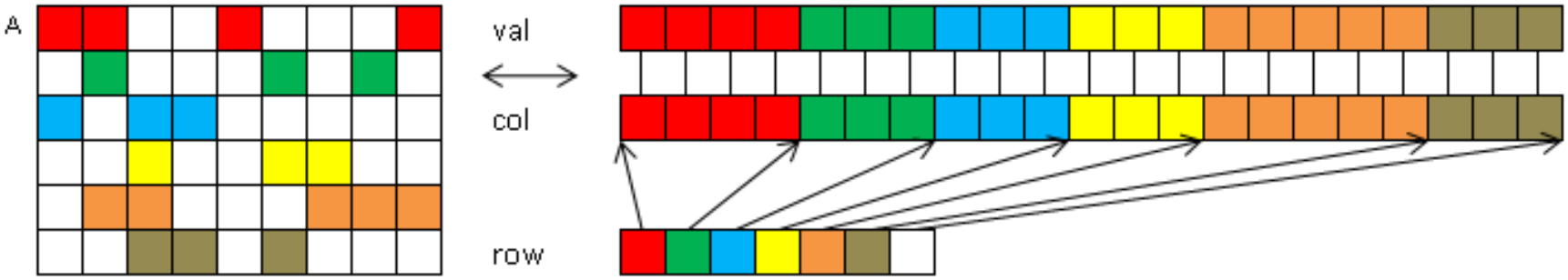


GMRES-Verfahren

- Iteratives, numerisches Verfahren zur Lösung großer, dünnbesetzter Gleichungssysteme
- Pro Iteration:
  - Mehrere Skalarprodukte
  - Dominierend:
    Matrix-Vektor-Multiplikation (SpMxV)

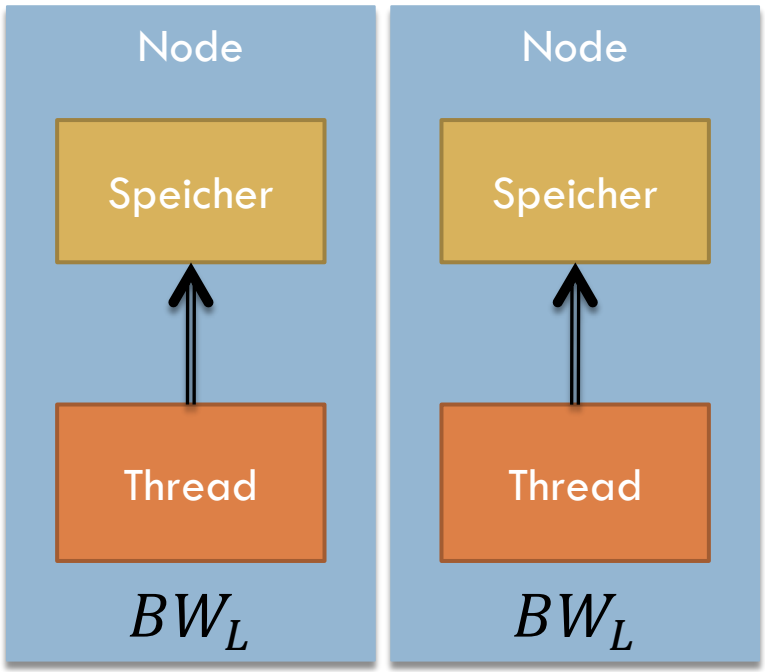
Matrix-Vektor-Multiplikation

- Multiplikation dünnbesetzter Matrix mit Vektor
- $y = A \times x$
- Matrix liegt zeilenweise im CRS Format vor

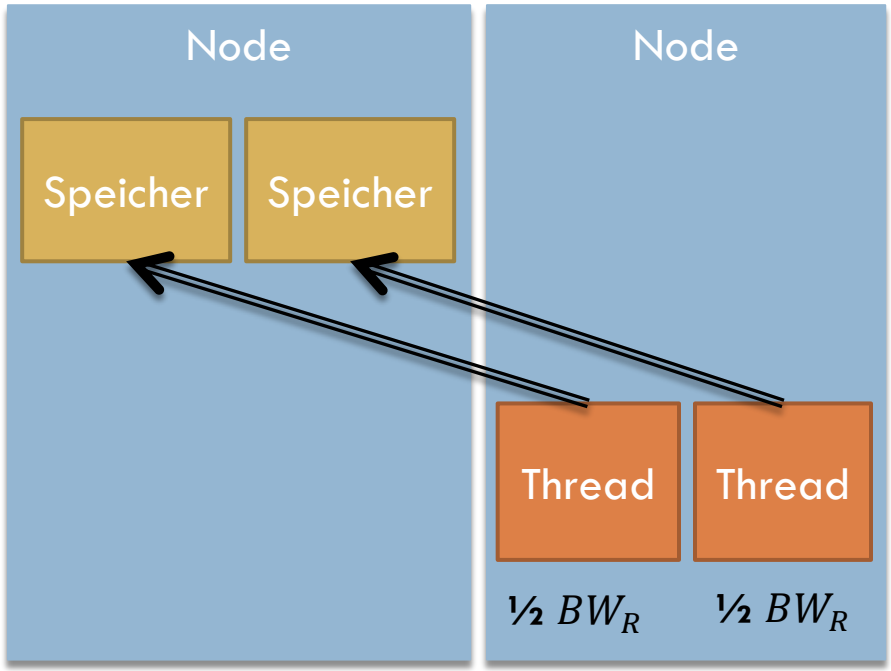


Motivation für Datenverteilung

- $BW(t) = P_D(t) \cdot BW_L + \left(1 - P_D(t)\right) \cdot BW_R$



$= 2\,BW_L$



$= BW_R \leq BW_L$

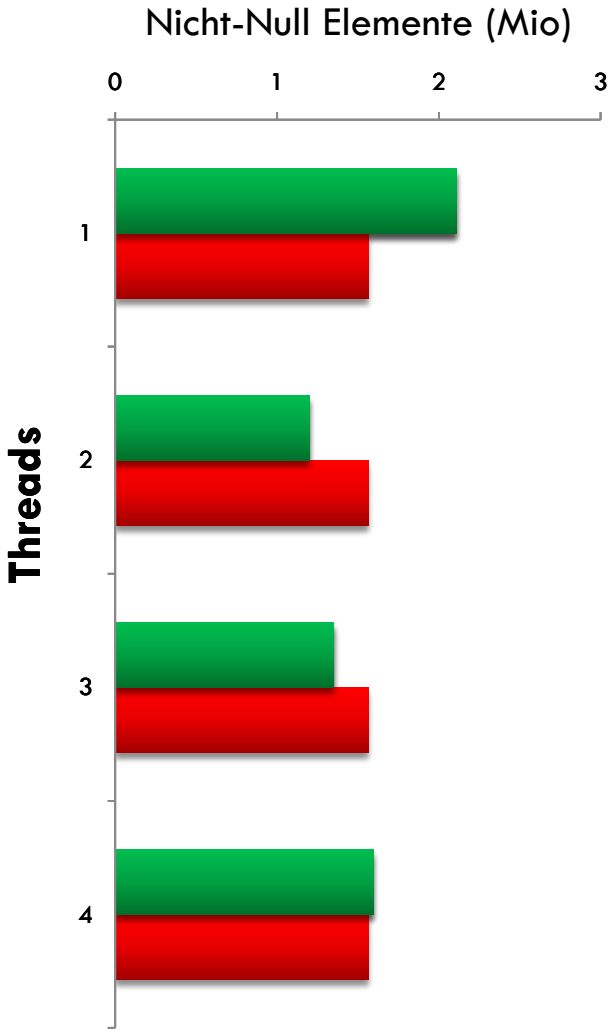
SpMxV – Datenverteilung

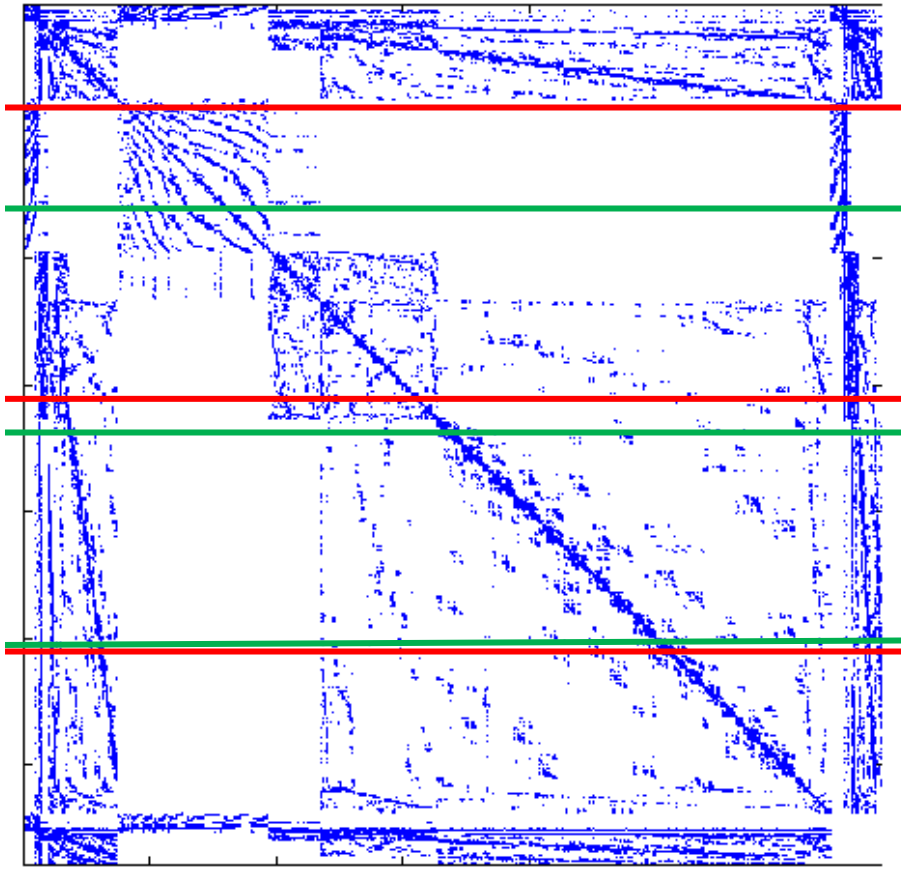
- Iterationen müssen gleichmäßig auf Threads verteilt werden
- Schwierig: Struktur der Matrix erst zur Laufzeit bekannt

Daher:

- Festes Binden von Threads auf Prozessoren
- Verteilung der Matrix zur Laufzeit auf diese Threads

Verteilung auf 4 Threads





67 385x67 385, Nicht-Null: 6 253 281

Chunked-Verteilung der Matrix

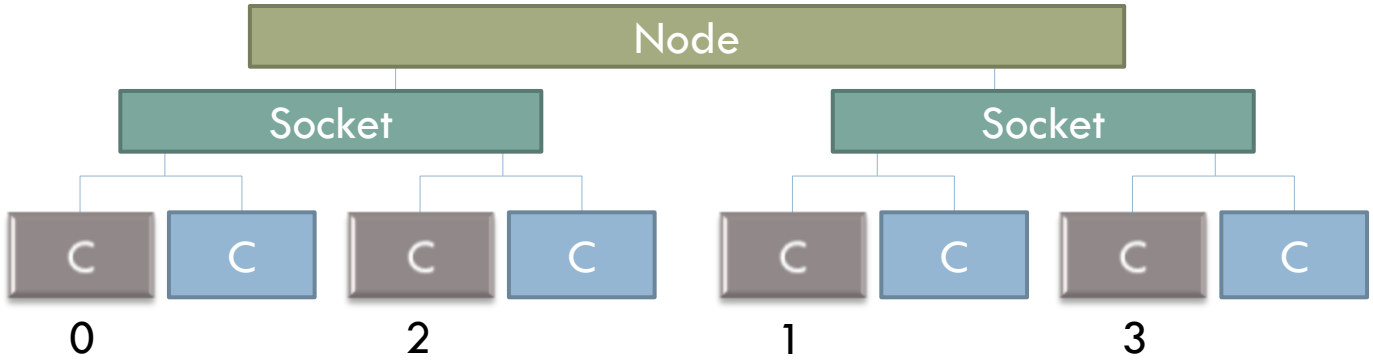
- Verteilung der Matrix berechnen
- Verteilung – Chunks – speichern
- Verteilung bei SpMxV verwenden
  - Loop-Schedule

Integration in laperf

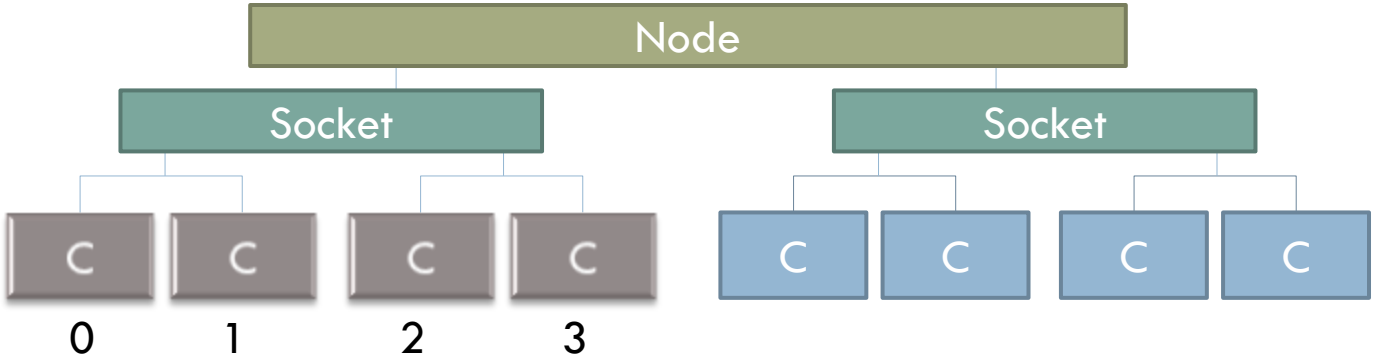
Thread Binding

- laperf um einfaches Laufzeitsystem erweitert
- Initialisierung beim Programmanfang
- Binden von Threads auf verschiedene Arten:

Scatter



Compact



Speicherverteilung

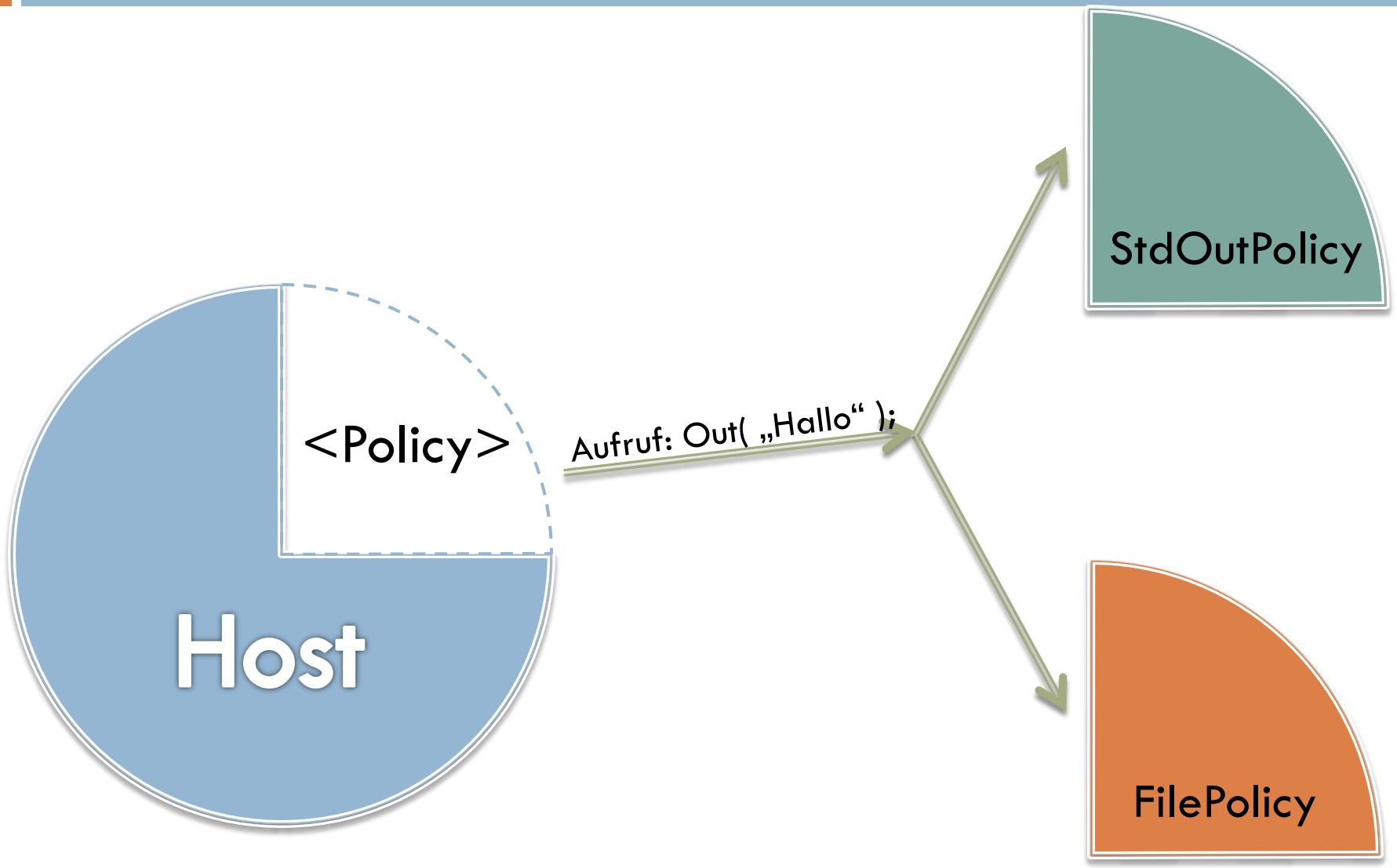
- laperf Datentypen verwenden STL Container

```
std::vector<double> data_;
```

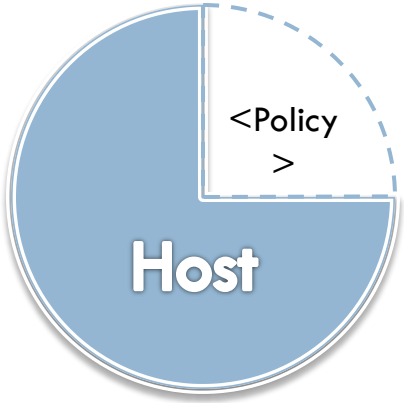
- STL Container akzeptieren einen **Allocator**

```
std::vector<double,std::allocator> data_;
```

Policy Based Class Design



```
template<class Policy>
class Host : protected Policy {
       void TueEtwas {
              Out( “Hallo Welt” );
       }
}
```




```
class StdOutPolicy {
       void Out( string s ) {
              cout << s << endl;
       }
}
```

StdOutPolicy

```
class FilePolicy {
       void Out( string s ) {
              fwrite( s );
       }
}
```

FilePolicy

MemoryPolicies

Implementierte Memory Policies:

- Default: Keine Speicherverteilung
- Distributed: Zeilenweise Round-Robin Verteilung
- Chunked: Anzahl der Nicht-Null Einträge pro Thread möglichst gleich gross halten

Beispiel:

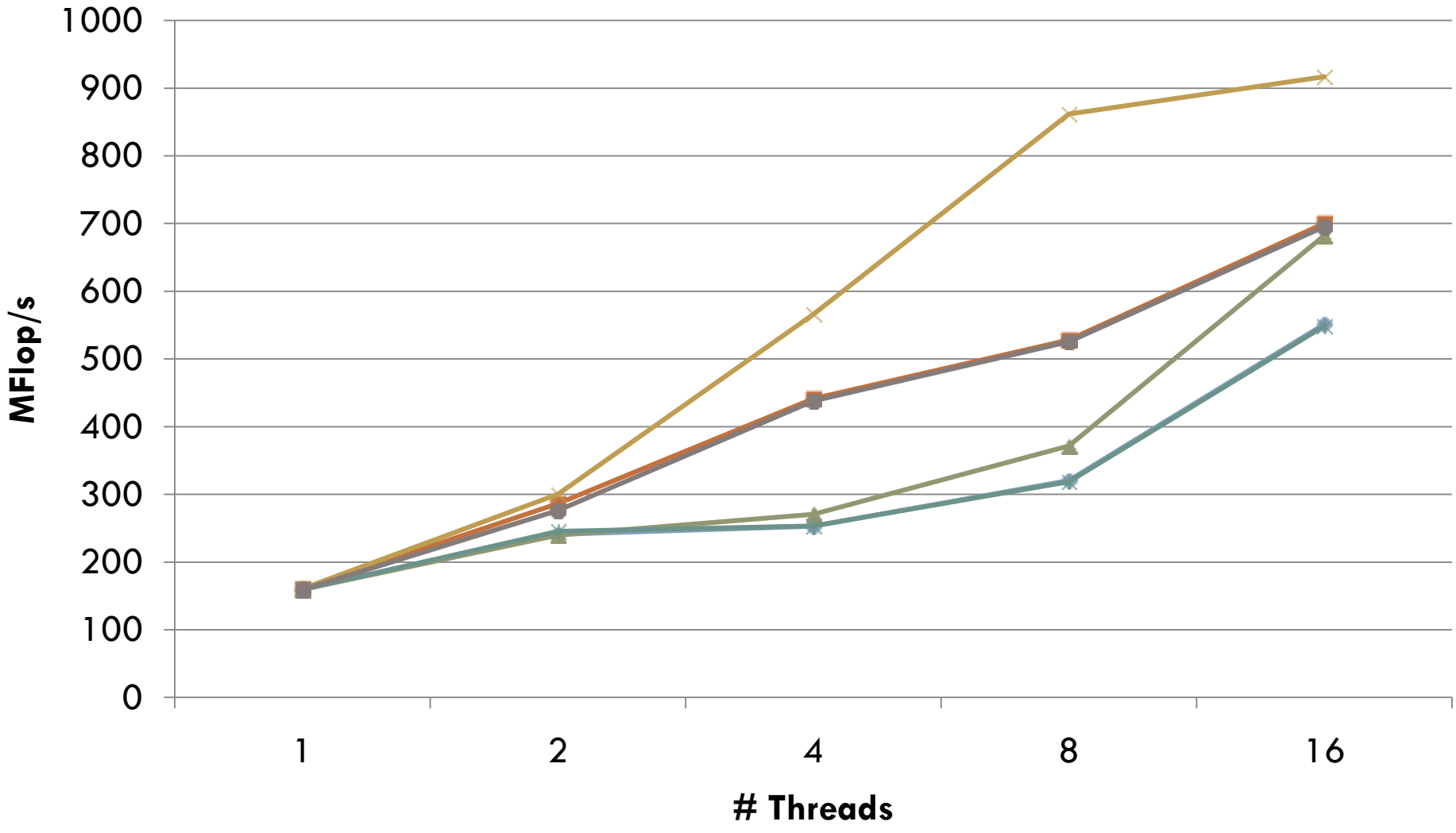
```
laperf::matrix_crs<double,
    MatrixMemoryPolicy::Chunked> m;
```

Auswertung

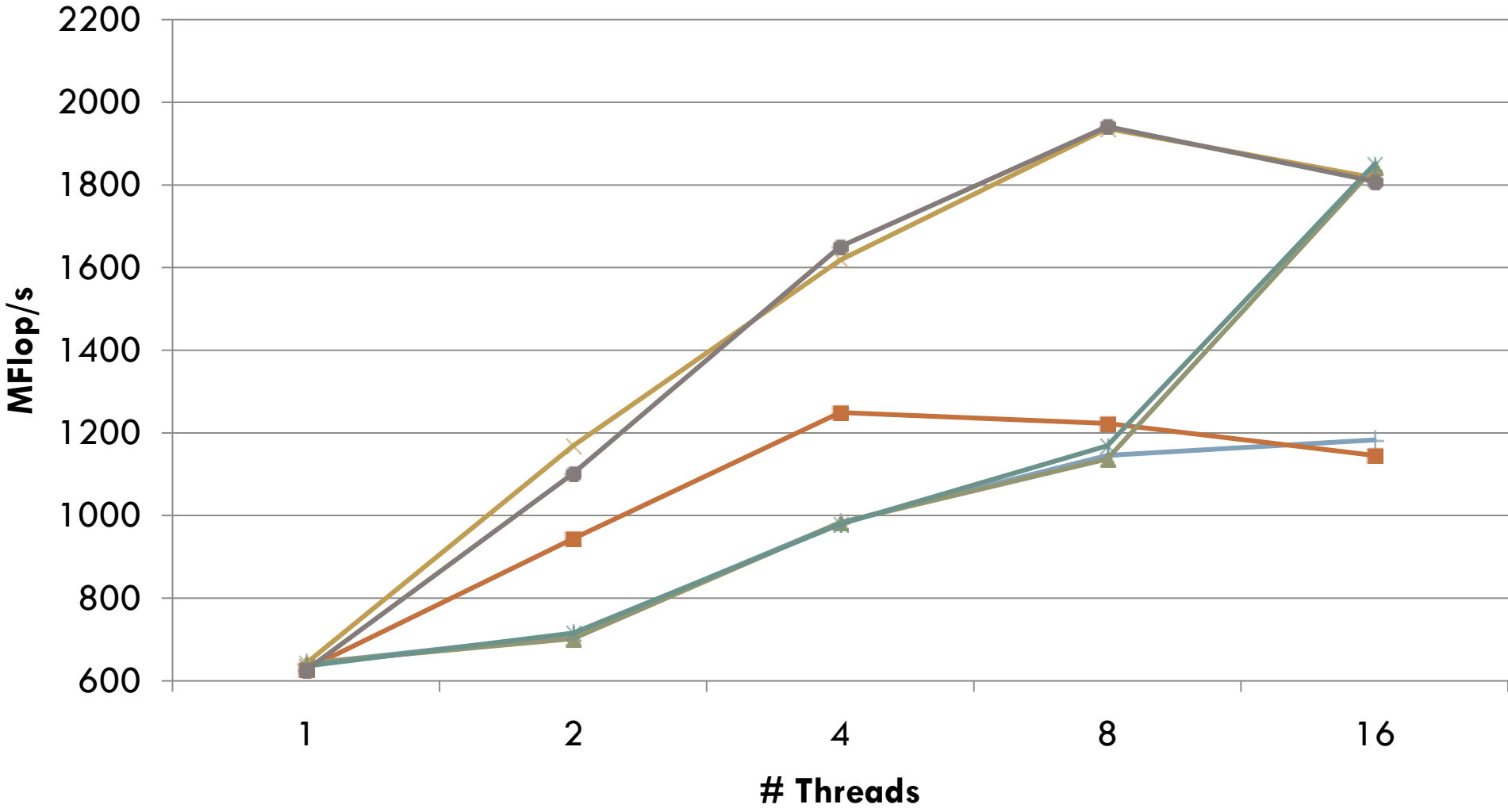
Verwendete Maschinen

- Intel Dunnington (SF X4450)
  - 4 Sockel, Hexacore Xeon Prozessoren
  - Uniform Memory Access
- Intel Nehalem (SF X4170)
  - 2 Sockel, Quadcore Xeon, 2 HW Threads/Core
  - **2 NUMA Nodes**
- AMD Barcelona (IBM LS42)
  - 4 Sockel, Quadcore AMD Opteron
  - **4 NUMA Nodes**

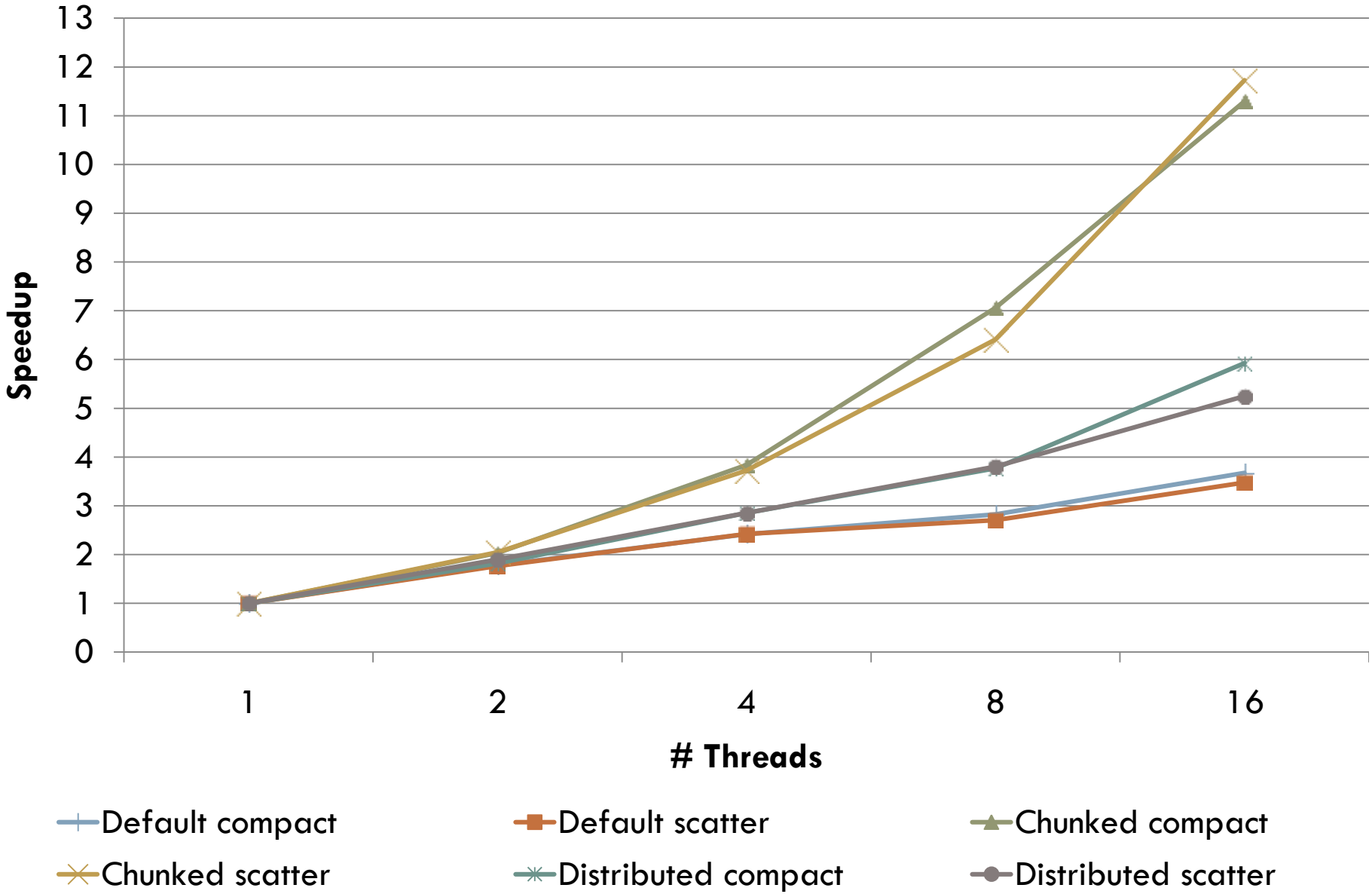
SpMxV – Intel Dunnington



SpMxV – Intel Nehalem



GMRES – AMD Barcelona



Paralleler Code (CG-Style)

```
laperf::matrix_crs<double,
  MatrixMemoryPolicy::Chunked> A(rows, cols, nonzeros);
laperf::vector<double,OMPInternPar> q(n), p(n), r(n);

[…]
for (int i = 1; i <= max_iter; ++i)
{
  […]
  q = A * p;
  double alpha = rho / (p*q);
  x += alpha * p;
  r -= alpha * q;
  […]
}
```

Zusammenfassung

Zusammenfassung

- Parallelisierung kann effizient gekapselt werden
- *High-Level* Interface für ccNUMA Optimierungen
  - Möglichkeit, dem Compiler mehr Informationen mitzuteilen
- Adaptierte C++ Konzepte
  - Policy Based Class Design
  - Allokatoren

Ausblick

- Verteilung weiter optimieren durch Einsatz von Hardware Countern
- Mehr *Intelligenz* in Laufzeitsystem


Fragen?

VIELEN DANK!

laperf – Grundlagen

Problem: Temporäre Kopien

```
laperf::vector<double> x, a, b;
x = (a * 2.0) + b;
```

Ideale Auswertung:

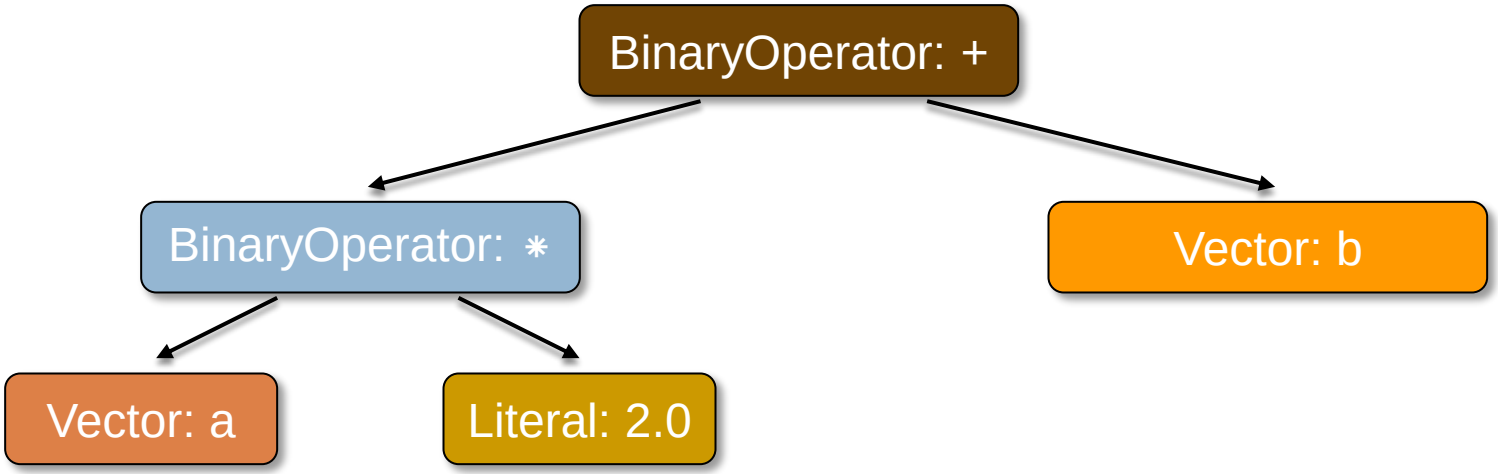
```
for( int i = 0; i < dim; ++i )
  x[i] = a[i] * 2.0 + b[i];
```

Aber:

```
laperf::vector<double> _t1 = operator*(a,2.0);
laperf::vector<double> _t2 = operator+(_t1,b);
x.operator=(_t2);
```

Template Expressions

```
x = (a * 2.0) + b;
```




```
LB<OpAdd, LB<OpMul,vector,double>, vector>
expr( LB<OpAdd,LB<OpMul, vector, double>, vector>(
     LB<OpMul, vector, double>(a,2.0), b
   )
);
```

```
LB<OpAdd, LB<OpMul,vector,double>, vector>
expr( LB<OpAdd,LB<OpMul, vector, double>, vector>(
    LB<OpMul, vector, double>(a,2.0), b
  )
);
```

Template Ausdruck auswerten: `operator=`

```
template<typename TExpr>
vector::operator=( TExpr expr ) {
  for( size_t i = 0; i < dim; ++i )
    this[i] = expr[i];
}
```

Inlining:

```
#pragma omp parallel for
for( size_t i = 0; i < dim; ++i )
  x[i] = a[i] * 2.0 + b[i];
```

SpMxV – Code

```
1 for( int i = 0; i < num_rows; ++i )
2 {
3   for( int r = row[ i ];
4        r < row[ i+1 ]; ++r )
5   {
6     y[i] += val[ r ] * x[ col[ r ] ];
7   }
8 }
```

Verwendete Matrix

- 275 990x275 990
- 26 930 841
  Nicht-Null Elemente