Shared-Memory Parallelisierung von C++ Programmen

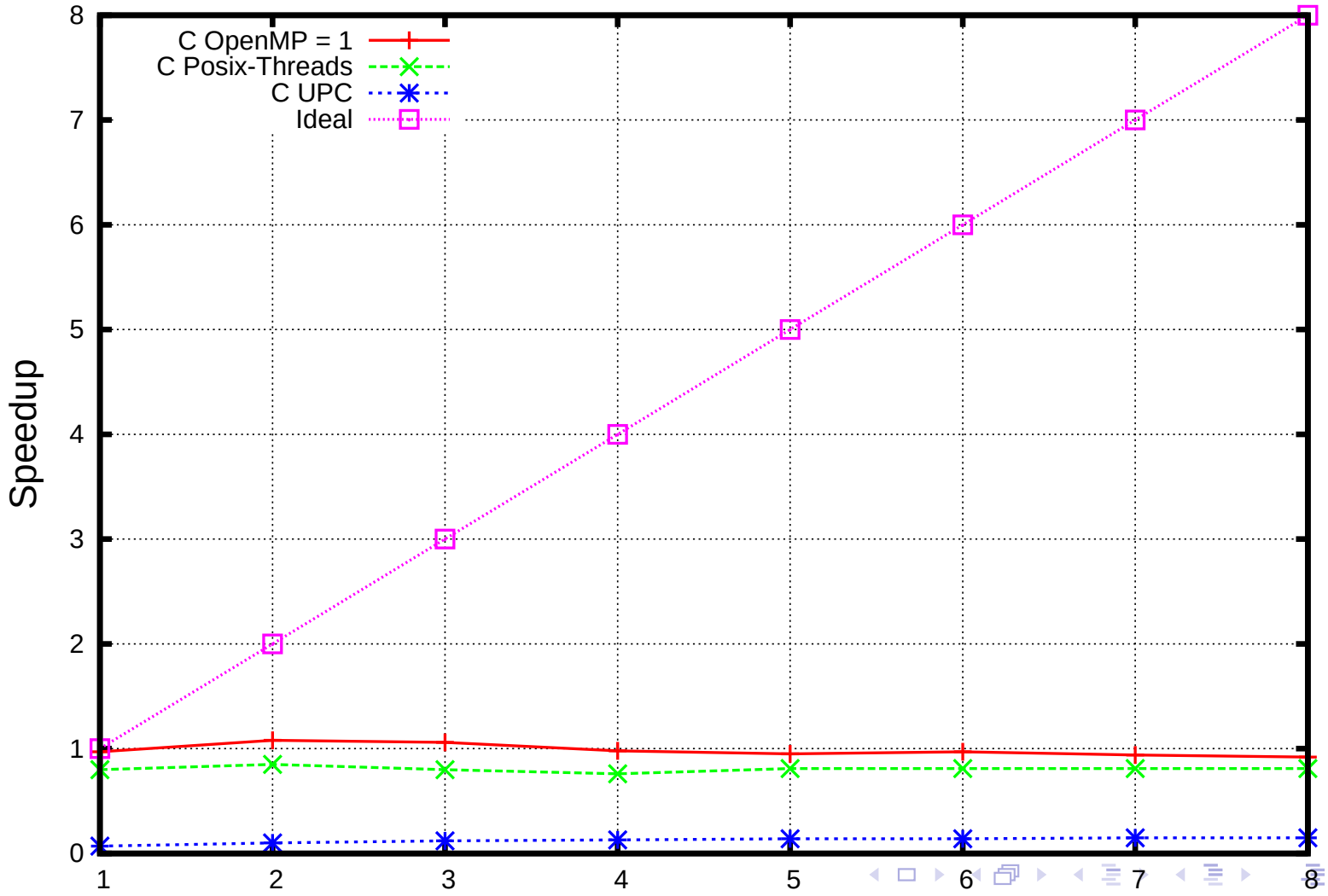
Christian Terboven

9. Februar 2006

Übersicht

- Verbreitete Parallelisierungstechniken für Shared-Memory:
  - *OpenMP*: Direktiven-gesteuerte Shared-Memory Parallelisierung für C, C++ und FORTRAN.
  - *Posix-Threads*: Shared-Memory (System-)Bibliothek mit umfangreicher Funktionalität, sprachunabhängig.
  - *UPC*: Übermenge von C99 mit Distributed-Shared-Memory Modell, kann mit C++ kombiniert werden.
  - *MPI-2*: wurde hier nicht betrachtet.
- Betrachtung der Parallelisierung des STREAM-Benchmarks und der DROPS Programmkernel.
- Vergleichskriterien:
  - Vorgehensweise, Tool-Unterstützung.
  - Performance und Aufwand.
  - Performance nach Tuning.

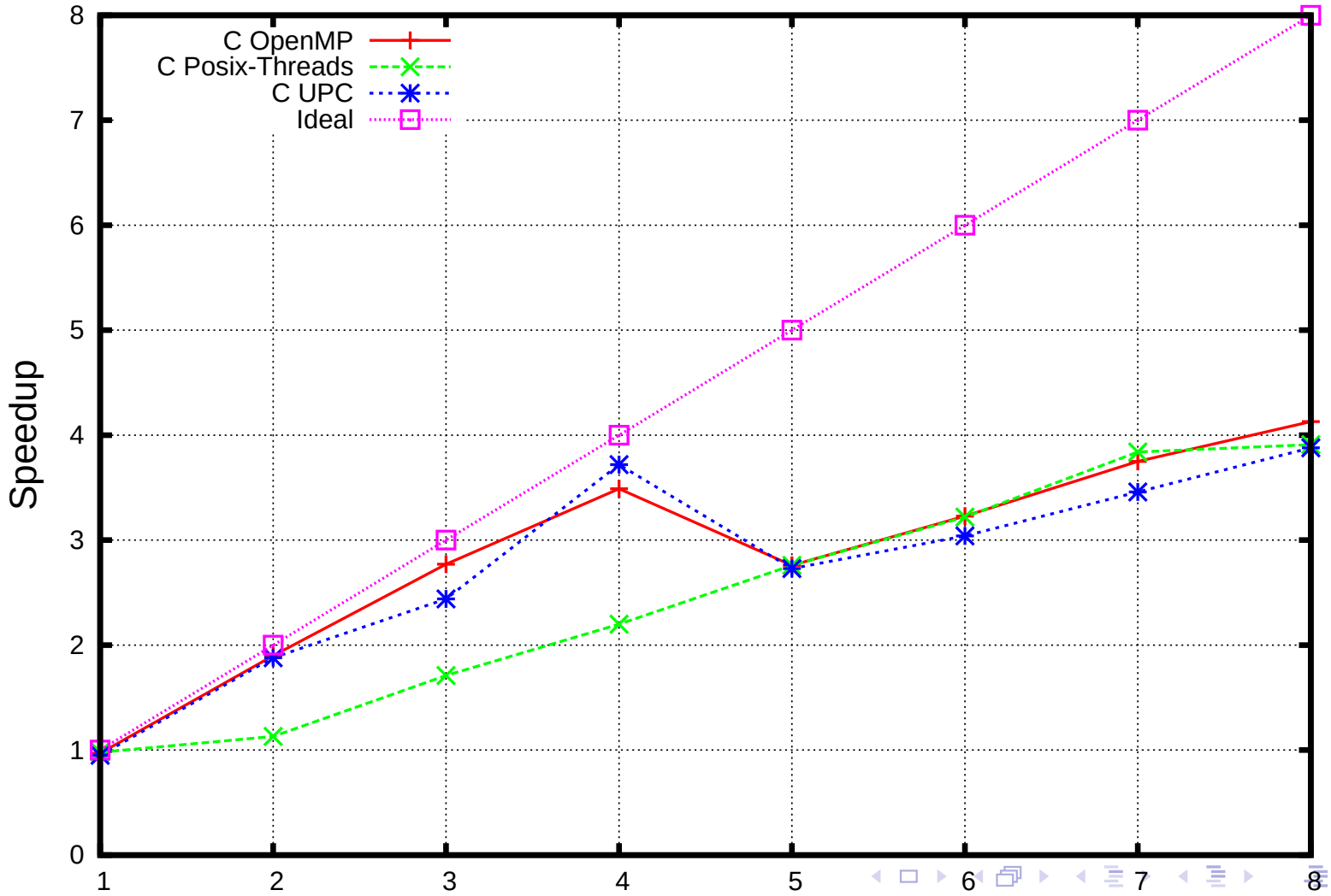
Performance: STREAM-Benchmark



Aufwand für Tuning

- Opteron-Plattform hat ccNUMA-Eigenschaften: Datenlokalität berücksichtigen!
- OpenMP: Initialisierungsschleife mit gleicher (paralleler) Arbeitsverteilung wie die Berechnung.
- Posix-Threads:
  - Initialisierungsschleife.
  - Thread-Pool zur Vermeidung des Aufwandes der Erstellung und Beendigung von Threads.
  - *CPU-Binding* der Threads.
- UPC:
  - Datenlokalität ist kein Problem.
  - Compiler erzeugt „schlechten“ Code.
  - Verschiedene manuelle Optimierungen im Code und im Compiler (beta) verwenden.

Performance: STREAM-Benchmark (tuned)



Zusammenfassung

- Alle drei Parallelisierungstechniken eignen sich für die Parallelisierung des STREAM-Benchmarks und DROPS.
- UPC bietet leistungsfähige Möglichkeiten der Datenplatzierung. Ad-Hoc Parallelisierung ggfs. aufwändig.
- Posix-Threads:
  - Hohe Flexibilität, auch für die Programmierung irregulärer Codes geeignet.
  - Anpassung der Arbeitsverteilung ist aufwändig und muss manuell programmiert werden.
- OpenMP:
  - Schnellste Entwicklung und größter Benutzerkomfort.
  - Nur minimale Codeänderungen notwendig, Möglichkeit zur seriellen Compilation besteht weiter.
  - Tool-Unterstützung, z.B. Test auf serielle Äquivalenz.

Thread-Safety

- Ein Code ist *thread-safe*, wenn er bei gleichzeitiger Ausführung durch mehrere Threads korrekt ist.
- Hier betrachtete STLs: Sun C++ und STLport (eng verwandt mit SGI STL), GNU C++ und Intels Modifikationen.
- Zwei Szenarien:
  - Mehrere Threads greifen (schreibend) auf eine Instanz eines Datentypes zu.
  - Mehrere Threads greifen (schreibend) auf mehrere Instanzen zu, nie mehr als ein Thread auf eine Instanz.
- Eine Funktion ist *reentrant*, wenn:
  - sie nur Variablen vom Stack benutzt,
  - sie nur von aktuellen Argumenten abhängt,
  - alle aufgerufenen Funktionen es auch sind.

Ergebnis

- Die betrachteten STLs bieten ausschließlich *reentrante* Funktionen an. In der Sun STL und in STLport enthalten lediglich spezialisierte Allokatoren statische Variablen, die über Locking geschützt werden.
- Zusammenfassung:
  - Ausschließlicher Lesezugriff ist sicher.
  - Gleichzeitiger (schreibender) Zugriff mehrerer Threads auf mehrere unterschiedliche Instanzen ist sicher.
  - Gleichzeitiger (schreibender) Zugriff mindestens eines Threads muss von der Applikation verwaltet werden.

Einleitung

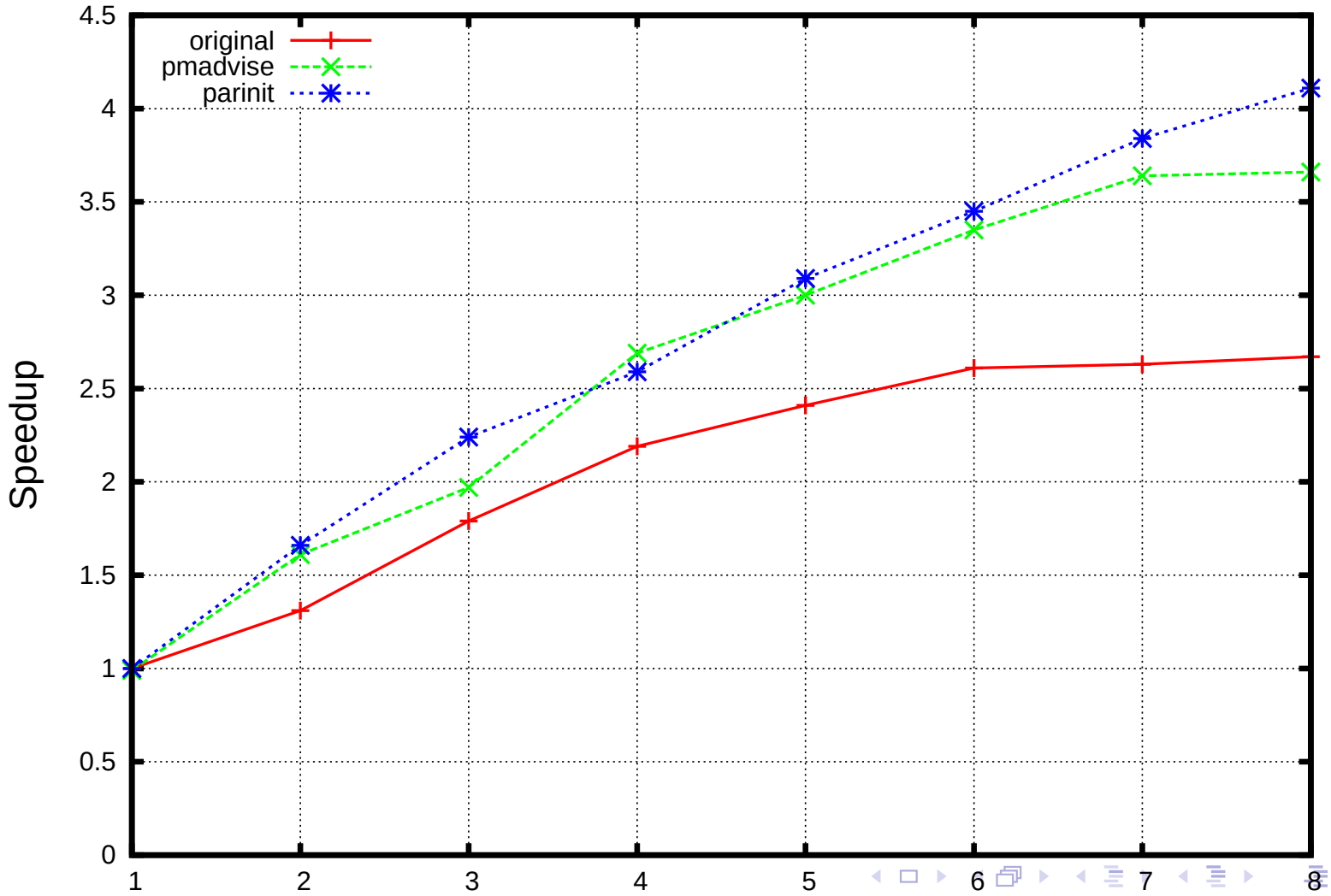
- Einige Datentypen haben sich als ungünstig zur Verwendung auf NUMA-Architekturen erwiesen, z.B: *std*::*valarray*.
- Eine Verteilung der Daten ist nach dem erstmaligen „Anfassen“ nicht mehr direkt möglich.
- Zwei Ansätze zur Verteilung der Daten:
  - Nutzung von Fähigkeiten des Betriebssystems (hier: Solaris) um Codemodifikationen zu minimieren.
  - Nutzung von C++ Sprachmitteln mit der Hoffnung auf Portabilität (*First-Touch*).
- Möglichkeiten zur Allokation dynamischen Speichers:
  - Zeiger,
  - Objekte aus Klassenbibliotheken (z.B. STL),
  - Allgemeine Klassen.

Zeiger

- Im Allgemeinen leicht an eine vorgegebene Datenverteilung anzupassen:
  - einzelner Zeiger auf Datenbereich kann explizit parallel initialisiert werden,
  - Array von Zeigern wird parallel verteilt angelegt.
- Beispielcode ADI: Poisson-Löser. Berechnung erfolgt auf der Variablen *gridPoint ** grid*.
- Betriebssystem: Benutzung von *pmadvise* mit Strategie *MADV_ACCESS_MANY*.
- C++ und OpenMP: Optimierung der Verteilung beim Anlegen des Feldes:

```
// Allocation of variable grid
#pragma omp parallel for
for (int i = 0; i < N; i++) {
  grid[i] = new gridPoint;
}
```

Performance: ADI mit Initialisierungstechniken



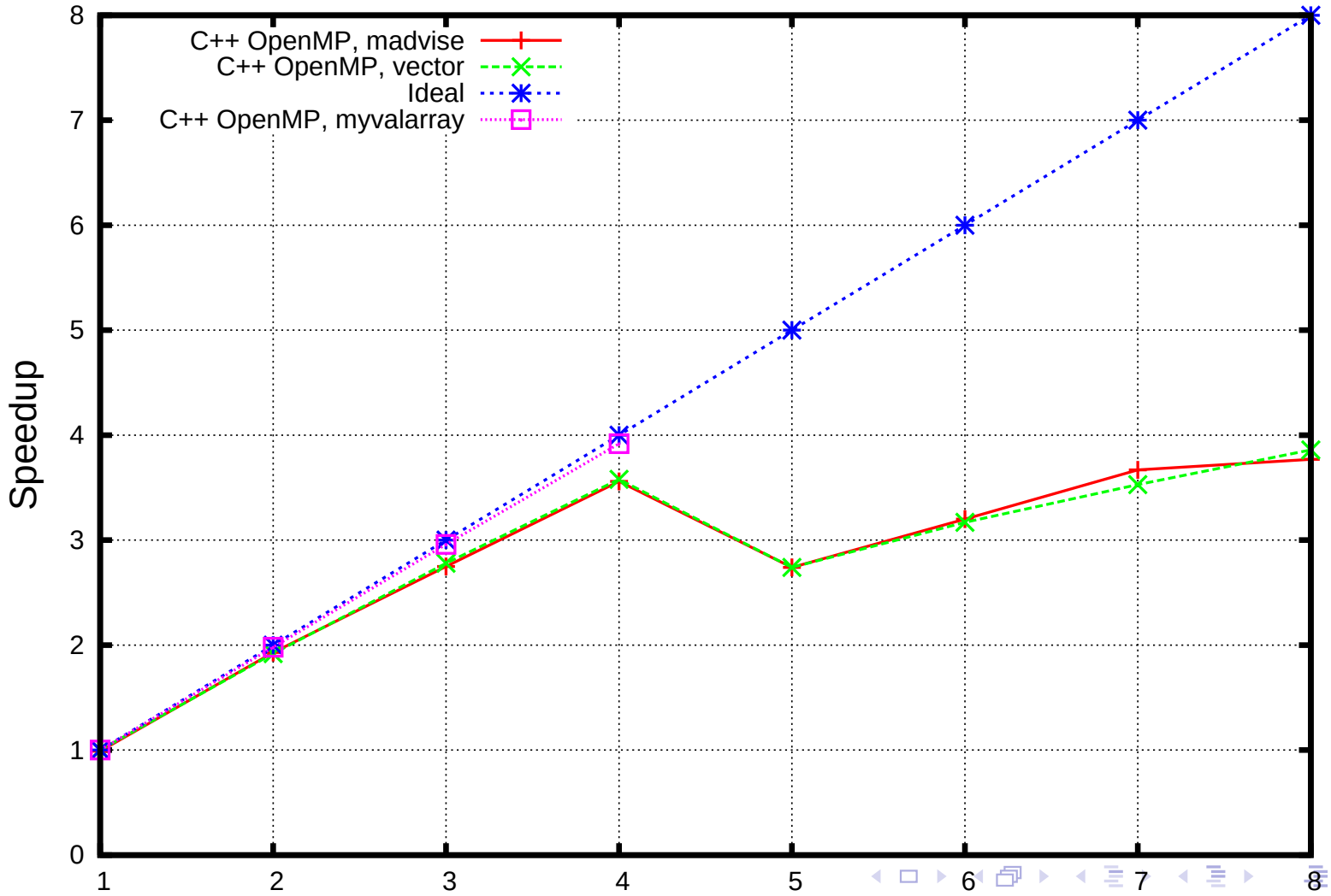
Objekte aus Klassenbibliotheken

- Falls Objekte keine Möglichkeit zur Beeinflussung der Speicherverwaltung anbieten:
  - Einsatz von Betriebssystemfunktionalität, z.B. *madvise*().
  - Modifikation des Objektes, z.B. im Konstruktor.
- *std*::*valarray* bietet keine Möglichkeiten über seine Schnittstelle. Eine Modifikation der Allokationstechnik wurde durchgeführt, ist aber nicht zwischen Compilern portabel.
- *std*::*vector* bietet in weiten Teilen die gleiche Schnittstelle, erlaubt aber zusätzlich die Angabe eines Allokators.
- Ein *Allokator* ist eine Klasse, die Aufgaben der Speicherverwaltung übernimmt, und eine von der STL vorgegebene Schnittstelle implementiert.
- Beim Austausch von *std*::*valarray* mit *std*::*vector* ist auf die Performance zu achten.

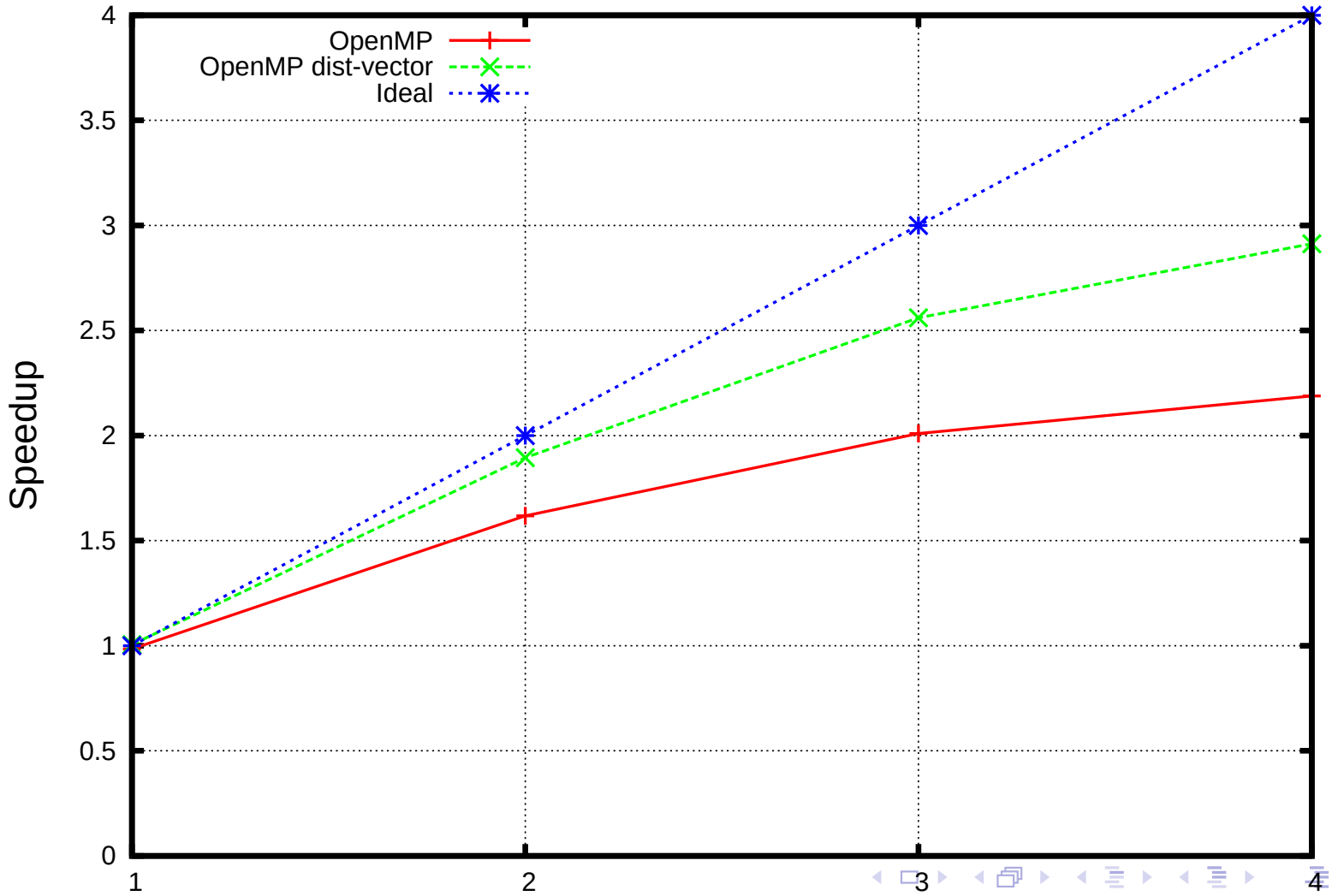
Allokator-Framework

- Framework zur Evaluation verschiedener Allokationstechniken:
  - *Allokator*: implementiert die von der STL geforderte Schnittstelle, verwendet *HeapManager*.
  - *HeapManager*: verwaltet Speicher, der vom *HeapLayer* allokiert wurde.
  - *HeapLayer*: allokiert physikalischen Speicher durch System- oder Bibliotheksaufrufe.
- *DistributedHeapAllocator*: implementiert einen Allokator, der Speicher direkt nach der Allokation mittels OpenMP parallel mit null beschreibt. Das Scheduling wird durch einen Templateparameter festgelegt.
- Der *DistributedHeapAllocator* kann mit *std*::*vector* verwendet werden.

Performance: STREAM-Benchmark mit Verteilung



Performance: Routine *y_Ax* aus DROPS (sparse MatVec)



Allgemeine Klassen

- Über ein *Mixin* kann die Speicherverwaltung einer Klasse modifiziert werden, wie in [Berger & Zorn + 01] vorgestellt.

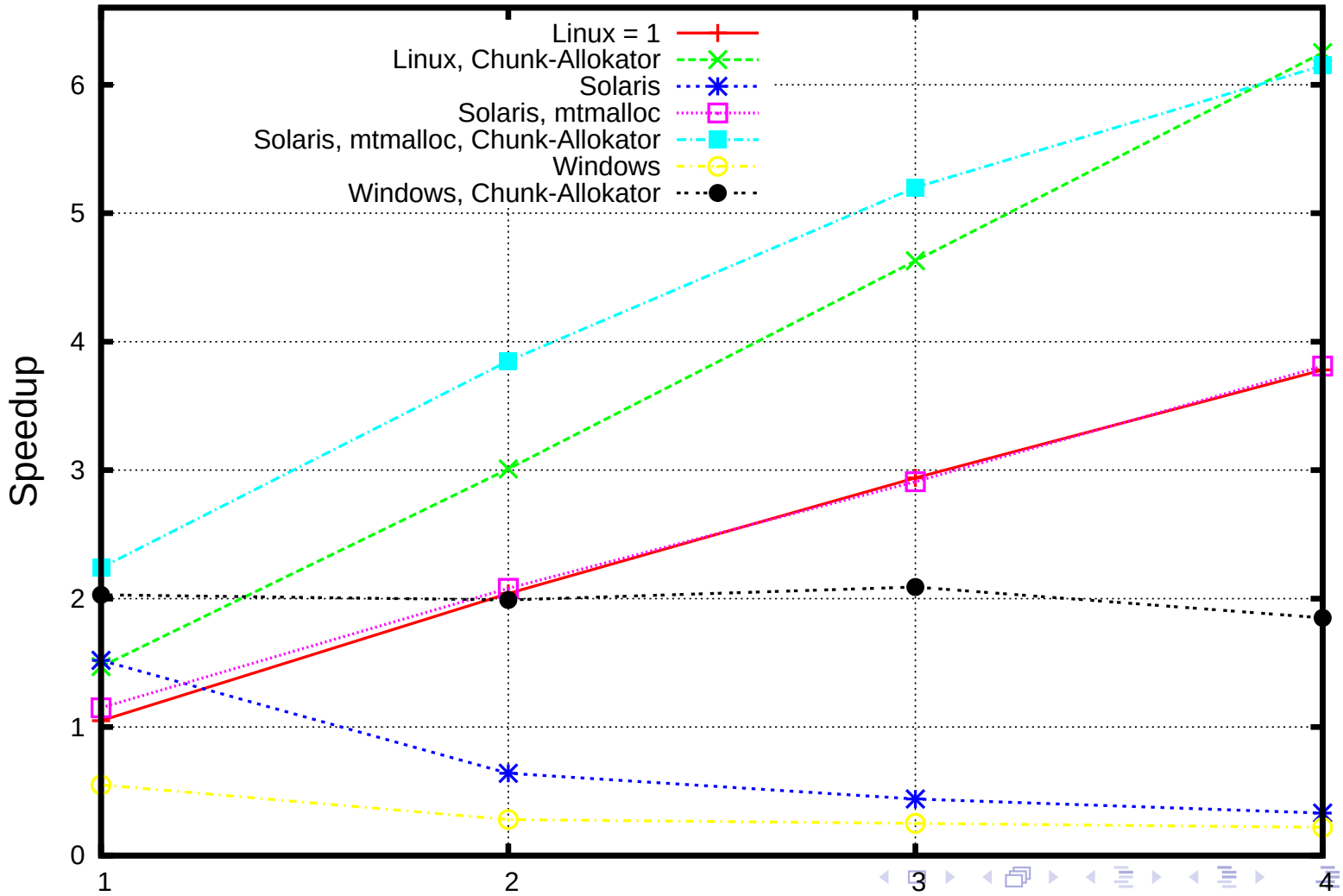
```
template<class Object, class HeapManager>
class PerClassHeapMixin : public Object {
 public:
  inline void * operator new (size_t sz) {
    return getHeap().malloc (sz);
  }
  static HeapManager & getHeap () {
    static HeapManager theHeap;
    return theHeap;
  }
[...]
};
```

- Die Allokation von Speicher wird vom *HeapManager* verwaltet.

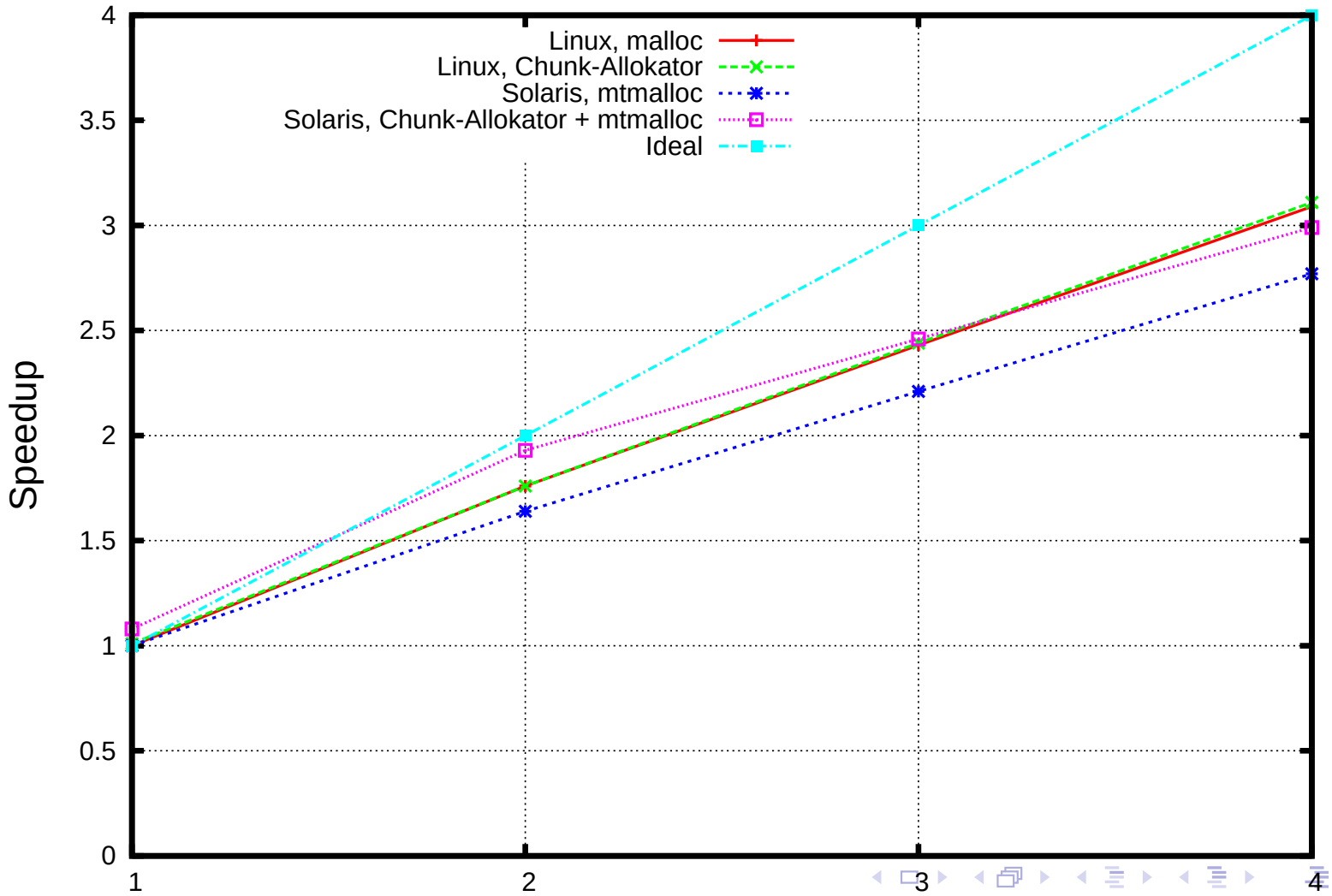
Übersicht

- Bei der Verwendung von Datenstrukturen, die viele kleine Objekte anlegen, wurden Probleme beim Speichermanagement unter Solaris und Windows festgestellt.
- Hier: Betrachtung von *std*::*map*, aber auch mit *std*::*list* und ähnlichen Datentypen können ähnliche Probleme auftreten.
- Unter Solaris werden spezielle Allokatoren zur effizienten seriellen Allokation verwendet.
- Lösungsansätze:
  - Verwendung einer Betriebssystem-nahen Bibliothek zur parallelen Allokation, z.B. *libmtmalloc*.
  - Modifikation der Speicherverwaltung seitens der Anwendung durch Anforderung größerer Blöcke.
- *std*::*map* erlaubt die Angabe eines Allokators.

Performance: MAP-Benchmark



Performance: Routine *SetupSystem*1 aus DROPS



Einleitung

- Parallelisierung eines High-Level Codes:
  - Interne Parallelisierung.
  - Externe Parallelisierung.
  - Verwendung in Parallelisierung (Thread-Safety).

```
PCG(const Mat& A, Vec& x,const Vec& b, ...)
{
    Vec p(n), z(n), q(n), r(n); Mat A(n, n);
    [...]
    for (int i = 1; i <= max_iter; ++i)
    {
        q = A * p;
        double alpha = rho / (p * q);
        x += alpha * p;
        r -= alpha * q;
        [...]
```

Objektorientierte Codes

- Es kann bereits aufwändig sein, seriell eine optimale Performance zu erreichen (*Template Expressions*).
- Interne Parallelisierung: abgeschlossene parallele Region innerhalb der Memberfunktionen.
  - Vorteil: Keine Änderung der Schnittstelle, korrekte Verwendung garantiert.
  - Nachteil: Aufwand beim Erzeugen und Beenden der Threads.
- Ebenfalls zur internen Parallelisierung zählen z.B. parallele Erweiterungen der STL. Diese können bei den betrachteten Codes aber nicht gewinnbringend eingesetzt werden.
- Externe Parallelisierung: parallele Region außerhalb der Memberfunktionen, innerhalb wird *Orphaning* eingesetzt.
  - Vorteil: Verringerung des Aufwandes des Threadmanagements.
  - Nachteil: Fehlerhafte Verwendung möglich.

Performance: objektorientierter PCG-Löser

- Intel C++ Compiler 9.0 auf Sun Fire V40z:

Version	**1**	**2**	**3**	**4**
Intern	219	108	83,2	65,5
Intern (temp. Vektor)	218	108	82,5	65,1
Extern	216	107	82,3	65
C-Code	216	107	82,3	65

- Sun C++ Studio10 Compiler auf Sun Fire E6900:

Version	**1**	**2**	**3**	**4**	**6**	**8**
Intern	375	185	156	112	68	42
Intern (temp. Vektor)	427	211	178	128	78	47
Extern	320	161	141	97	61	37
C-Code	315	157	138	95	57	35

Parallelisierung von nicht OpenMP konformen Schleifen

- Bei Verwendung von Zeigern oder Iteratoren (Kapselung eines Zeigers in der STL) sind die entstehenden Schleifen nicht von der kanonischen Form, wie sie OpenMP fordert.
- Wichtigste Forderung: Berechnung der Schleifeniterationen im Vorraus möglich. Dies würde auch bei Iteratoren (*distance*()) und Zeigern (Zeigerarithmetik) gehen.

Parallelisierung von nicht OpenMP konformen Schleifen

- Ansatz 1: Erstellung einer parallelisierbaren Schleife:

```
long l = 0, lSize = 0;
for (it = list1.begin(); it != list1.end(); it++)
  lSize++;
valarray<CComputeItem*> items(lSize);
for (it = list1.begin(); it != list1.end(); it++) {
  items[l] = &(*it); l++;
}
#pragma omp parallel for default(shared)
for (long l = 0; l < lSize; l++) {
  items[l]->compute();
}
```

Parallelisierung von nicht OpenMP konformen Schleifen

- Ansatz 2: Intels *Taskqueuing*:

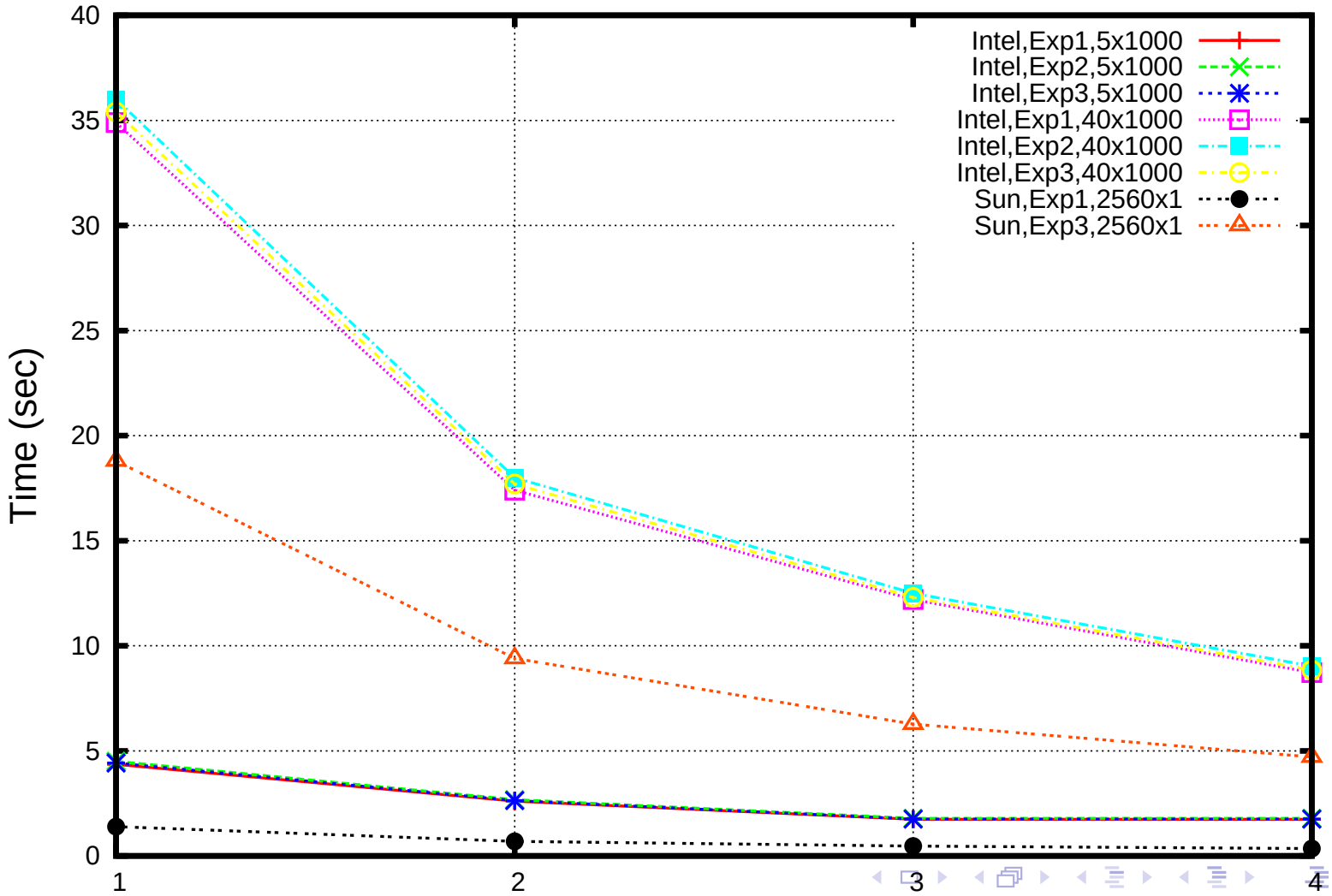
```
#pragma intel omp parallel taskq
{
for (it = list2.begin(); it != list2.end(); it++) {
#pragma intel omp task
{
  it->compute();
}
} // end for
} // end omp parallel
```

Parallelisierung von nicht OpenMP konformen Schleifen

- Ansatz 3: *single−nowait* Technik:

```
#pragma omp parallel private(it)
{
for (it = list3.begin(); it != list3.end(); it++) {
#pragma omp single nowait
{
   it->compute();
}
} // end for
} // end omp parallel
```

Performance: parallele Iteratorschleifen



Kritik an der OpenMP Spezifikation (1)

- Insgesamt war die Parallelisierung der betrachteten Anwendungsprogramme mit OpenMP erfolgreich.
- Bereits diskutiert: Schleifen mit Indexvariablen von anderem Typ als *Integer*.
- Schleifen über *size_t* werden ebenfalls nicht akzeptiert, da vorzeichenlos, aber häufig verwendet.
- Die Privatisierung von Membervariablen wird nicht unterstützt. Dies wäre angenehm beim *Chunk−Allokator* und bei der internen Parallelisierung.
- Eine Routine kann nicht erkennen, ob sie aus einem Worksharingkonstrukt aufgerufen wird. Dies wäre zur Garantierung der Korrektheit bei der externen Parallelisierung angenehm.

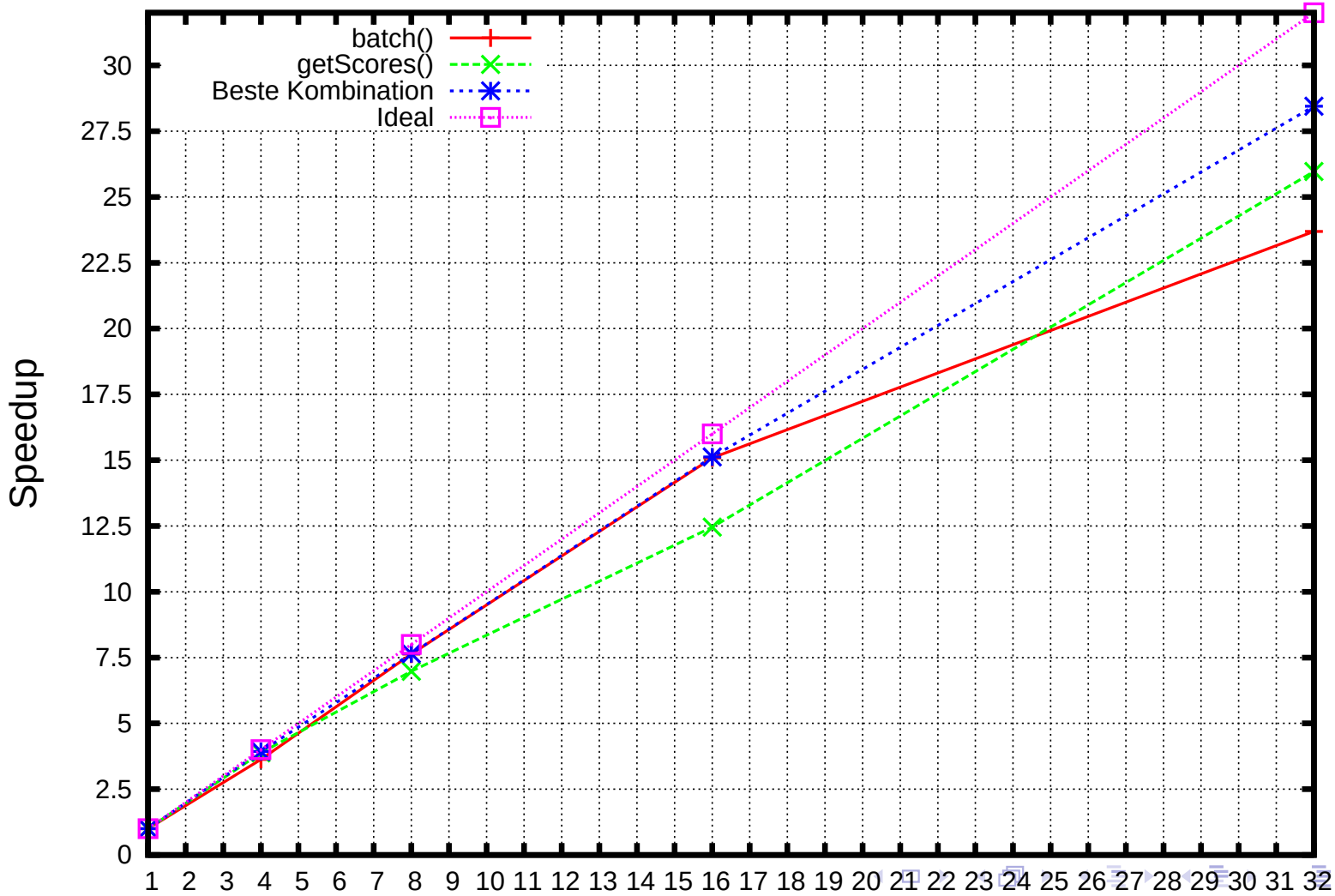
Kritik an der OpenMP Spezifikation (2)

- Die OpenMP Spezifikation geht nicht in allen Punkten präzise auf die Programmiersprache C++ ein.
- Privatisierung von Klassen:
  - OpenMP: Private Variablen sind nicht initialisiert.
  - C++: Für eine Instanz einer Klasse muss immer ein Konstruktor aufgerufen werden.
  - Der Intel C++ Compiler tut dies, der Sun C++ Compiler nicht. Dies führt zu schwer auffindbaren Fehlern.
- Viele Compiler haben Schwierigkeiten, C++ Konstrukte wie Exceptions, Templates, Vererbung im Zusammenhang mit OpenMP umzusetzen. Dies hat die Parallelisierung von DROPS gezeigt.

FIRE

- Das Image-Retrieval System FIRE wird am Lehrstuhl für Informatik 6 der RWTH Aachen von Thomas Deselaers und Daniel Keysers entwickelt.
- Ziele von FIRE:
  - Vergleich von Bildmerkmalen (Features)
  - Vergleich von Bildvergleichsmaßen (Distanzen)
  - Untersuchung der Korrelation von Features
- Die Parallelisierung wurde durch die objektorientierte Programmierung stark unterstützt (Aufwand: ca. 4 Wochen).
- Die Programmiersprache C++ hat die Abhängigkeitsanalyse deutlich vereinfacht.
- Die Flexibilität der Entwicklung durfte nicht eingeschränkt werden. Bei der OpenMP Parallelisierung wurden nur wenige Zeilen Code hinzugefügt.

FIRE


batch()
getScores()
Beste Kombination
Ideal
Speedup
0
2.5
5
7.5
10
12.5
15
17.5
20
22.5
25
27.5
30
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Zusammenfassung

- OpenMP lässt sich gut zur Parallelisierung von C++ Programmen einsetzen. Es bietet dem Programmierer den meisten Komfort im Vergleich mit Posix-Threads und UPC.
- Skalierbarkeit von OpenMP C++ Programmen ist nicht immer zwischen Hardwarearchitekturen, Betriebssystemen und Compilern portabel:
  - Opteron: NUMA-Architektur und Datenlokalität.
  - Solaris und Windows: Speicherverwaltung.
  - Effizienz der OpenMP Implementierung.
  - Diese Punkte können mit der aktuellen OpenMP Spezifikation nicht direkt adressiert werden.
  - Die C++ Programmiersprache bietet Lösungsansätze.
- Objektorientiertes Design kann ungewollte Abhängigkeiten im Code verhindern und die Abhängigkeitsanalyse erleichtern.

Ende

Vielen Dank für Ihre Aufmerksamkeit.