Cloud-Task-Parallelisierung in .NET: Jenseits der lokalen Prozessor-Power

Prof. Dr. Luc Bläser

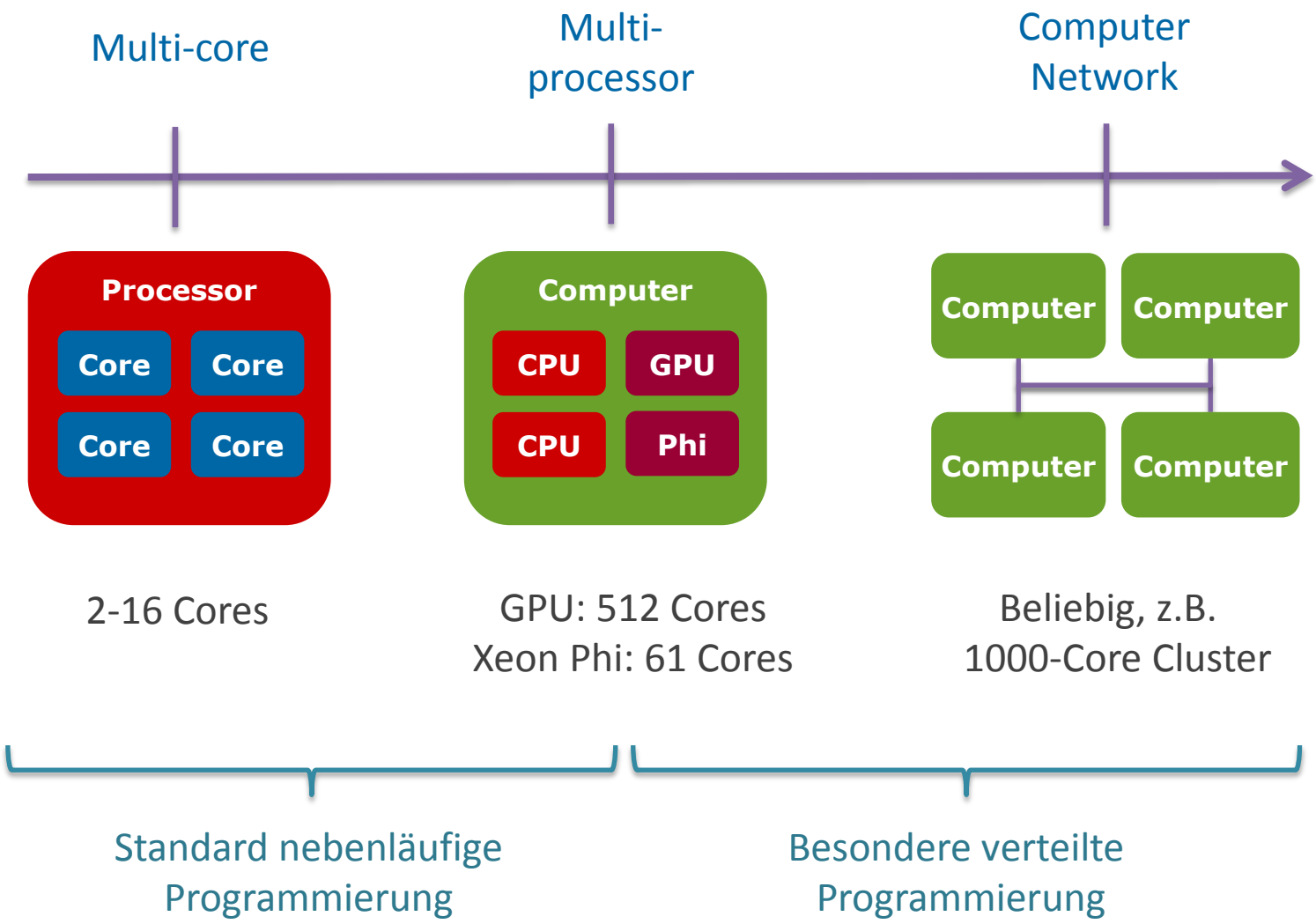
Hochschule für Technik Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUT
FÜR
SOFTWARE

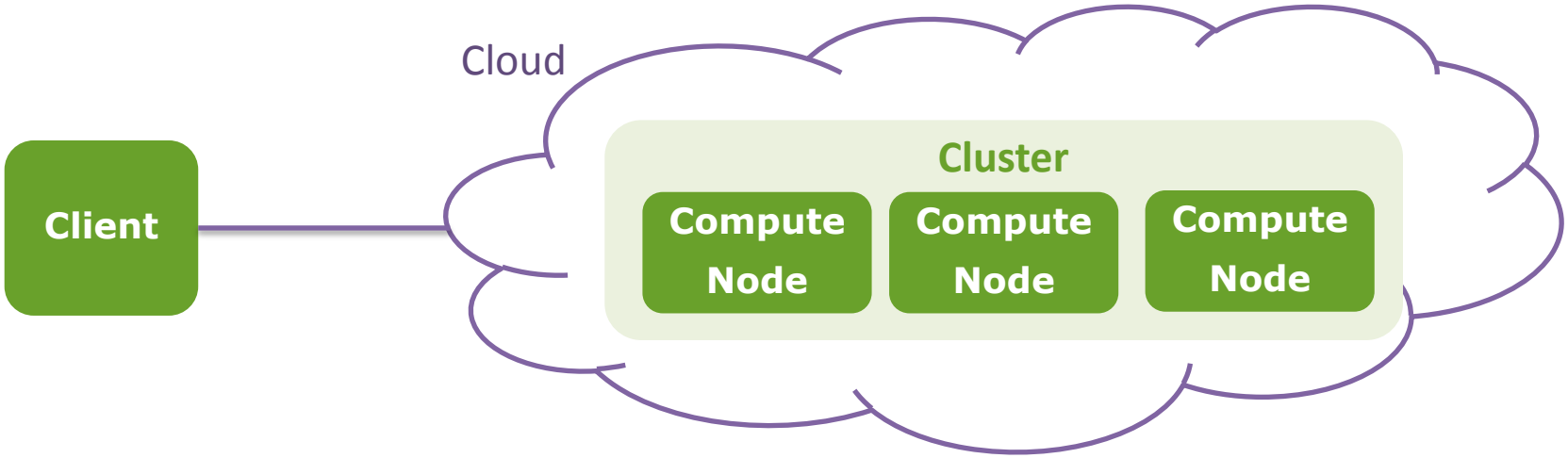
Parallel 2013, Karlsruhe
15. Mai 2013

Stufen der Parallelisierung



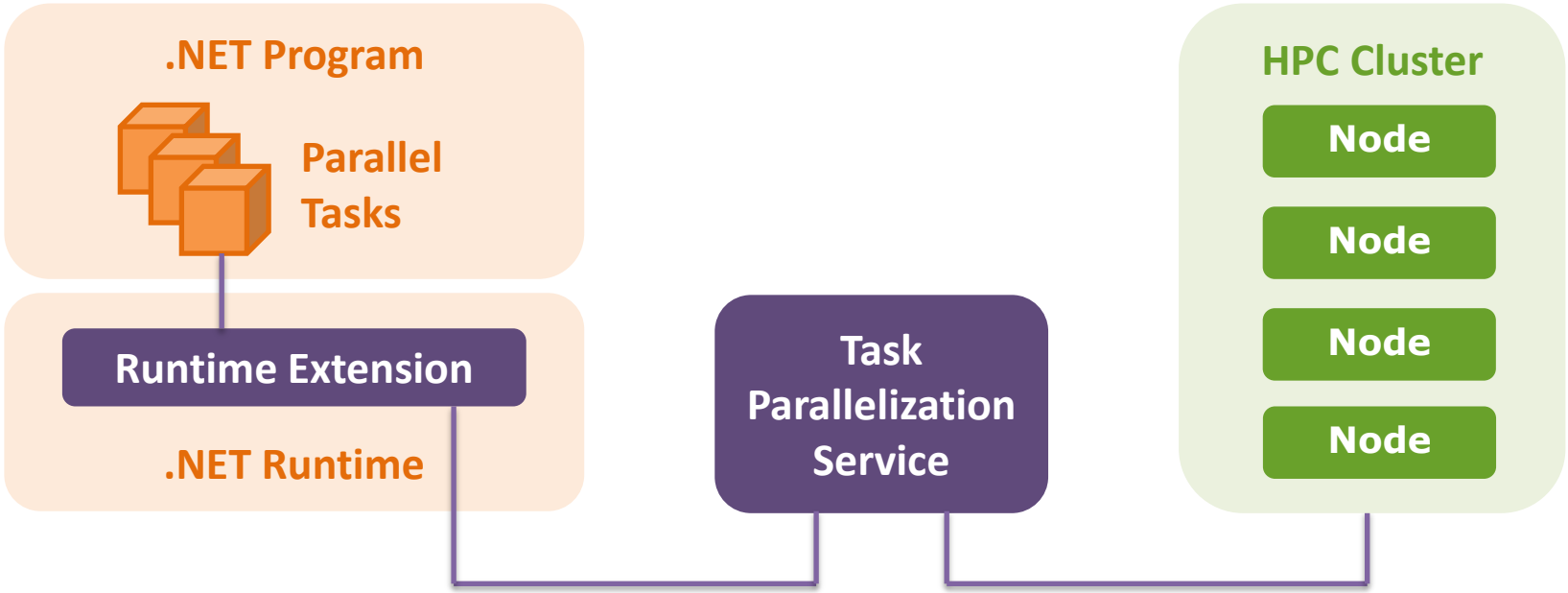
Ziel: Parallelisierung in der Cloud

- Entfernte Rechenpower lokal einbinden
  - Massive Parallelisierung in der Cloud
  - Z.B. auf Cluster mit vielen Multi-Cores Nodes
- Möglichst nahtlos und einfach nutzbar
  - Gleiches Programmiermodel wie für lokale Cores
  - Keinen explizite Teilung von Client/Server-Code



Cloud Task Parallelisierung in .NET

- Parallele Tasks in .NET programmieren
- Automatisch in Cloud senden und dort ausführen
- Cloud-Seite hat z.B. einen MS HPC Cluster



Überblick

- Programmiermodell
- Laufzeitsystem
- Experimentelle Resultate
- Schlussfolgerungen

Klassische .NET Task Parallelisierung

Mehrere Zahlen parallel faktorisieren

```
var taskList = new List<Task<long>>();
foreach (var number in inputs) {
  var task = Task.Factory.StartNew(
     () => _Factorize(number)
  );
  taskList.Add(task);
}

foreach (var task in taskList) {
  Console.WriteLine(task.Result);
}
```

TPL Task starten

Task Delegate (Lambda)

Ende abwarten vor Resultat

```
long _Factorize(long number) {
 for (long k = 2; k <= Math.Sqrt(number); k++) {
   if (number % k == 0) { return k; }
 }
 return number;
}
```

Neue Cloud-Task Parallelisierung

Analog zu TPL

Service angeben

```
var distribution = new Distribution(ServiceUri, Authorization);

var taskList = new List<DistributedTask<long>>();
foreach (var number in inputs) {
  var task = DistributedTask.New(
     () => _Factorize(number)
  );
  taskList.Add(task);
}

distribution.Start(taskList);

foreach (var task in taskList) {
  Console.WriteLine(task.Result);
}
```

Task erzeugen

Mehrere Tasks gebündelt starten

Library referenzieren: `HSR.CloudTaskParallelism.Client.Runtime`

Datenparallelität

Klassische .NET Parallelisierung

```
Parallel.For(0, inputs.Length, (i) => {
  outputs[i] = _Factorize(inputs[i]);
});
```

Neue Cloud Task Parallelisierung

```
distribution.ParallelFor(0, inputs.Length, (i) => {
  outputs[i] = _Factorize(inputs[i]);
});
```

Distributed Tasks

- Sehr ähnlich zu TPL
  - Lediglich Import einer Library: Kein Compile-Schritt
- Gebündelter Task-Start
  - Minimierung der Netz-Roundtrips
- Task als .NET Delegate/Lambda
  - Allgemeines Programmiermodell
- Tasks müssen unabhängig sein
  - Disjunkte Fields/Array Element-Zugriffe (ausser Read-Only)

Task Unabhängigkeit

- Kein Informationsfluss
  - Zwischen gleichzeitig laufenden Distributed Tasks
  - Zwischen laufenden Distributed Tasks und .NET Umgebung
- Isolationsprinzipien
  - Gemeinsam zugegriffener Speicher nur lesend
  - Sonst disjunkte Speicherzugriffe
  - Granularität: Field und Array Elemente
- Vereinfacht Verteilung
  - Gilt ebenso für nicht-synchronisierte .NET Tasks
  - Write/Write Konflikte werden vom Laufzeitsystem erkannt

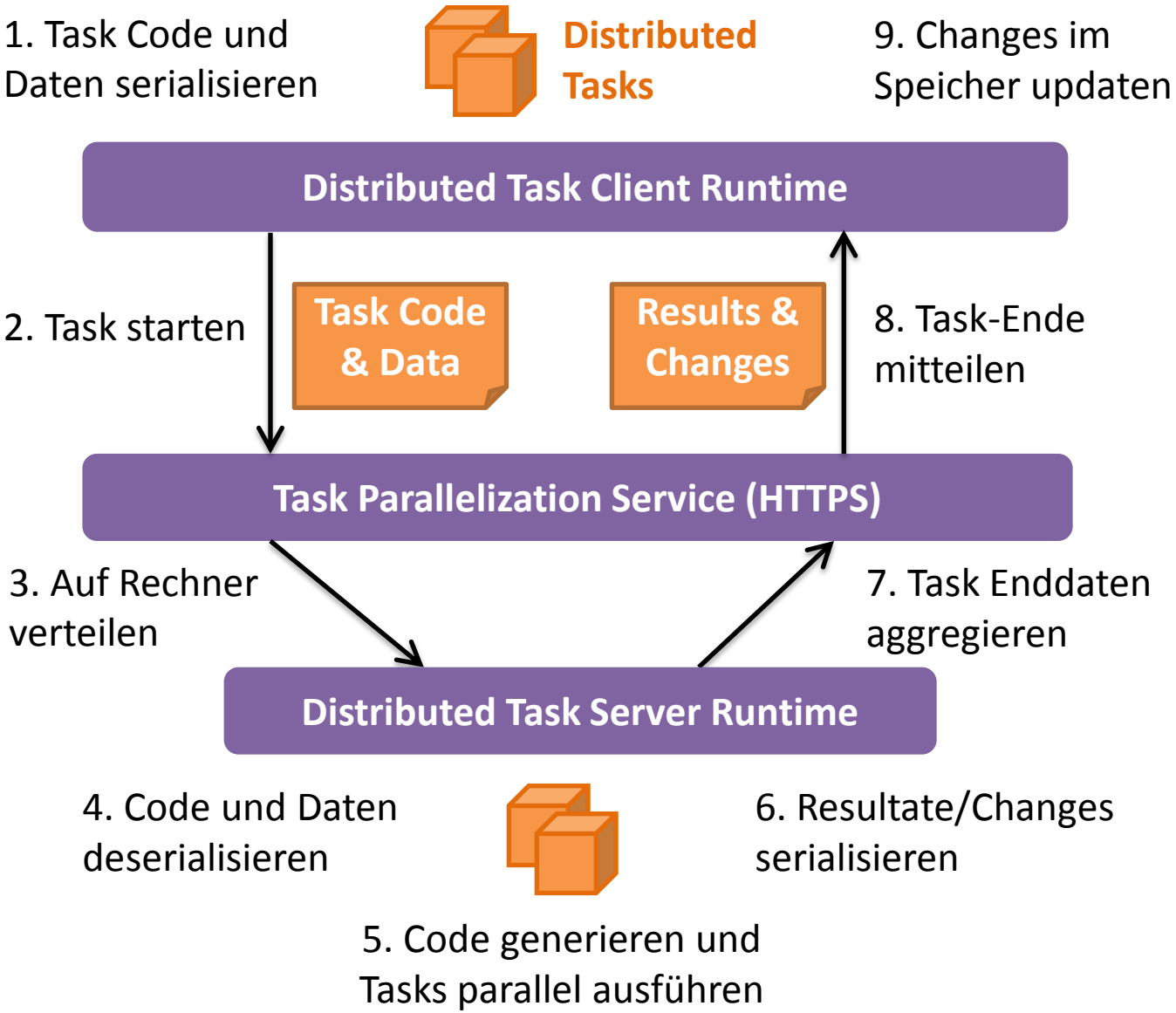
Distributed Task API

- Starten von Distributed Tasks
  - `distribution.Start(taskSet)`
- Warten auf Beendigung
  - `distribution.Await(taskSet)`
- Starten mit Warte-Barriere
  - `distribution.Invoke(taskSet)`
- `ParallelFor`, `ParallelForEach`

Einschränkungen

- By Design
  - Keine Interaktion / Synchronisation unter Tasks
  - Kein Task-Starts in Tasks
  - Kein I/O oder Systemcalls in Tasks (Sicherheitsgründe)
  - Kein Unmanaged Code (Sicherheitsgründe)
- By Implementation
  - Bestimmte Sprachfeatures noch nicht unterstützt
  - Laufzeitsystem wird stetig weiter ausgebaut
  - Genaue Liste später

Laufzeitsystem



Task Serialisierung

- Potentiell ausführbarer Task-Code
  - Konservative Code-Analyse
    - Ausgehend von Task Delegate
    - Direkt und indirekt aufrufbare Methoden
    - Potentiell verwendete Klassen und Fields
- Potentiell verwendete Task-Daten
  - Partieller Heap Snapshot
    - Graph erreichbarer Objekte mit zugreifbaren Fields
    - Zugreifbare statische Fields/Konstanten
    - Konsistenz wegen Task Isolation

Task Updates/Results

- Rückgabe vom Server bei Task Ende
  - Task Delegate Rückgabewert
  - Änderungen an Objekten/statischen Daten
    - Field Updates
    - Array Element Updates
  - Neu erzeugte Objekte
- In-Place Updates bei Client
  - Auf entsprechende Objekte des Input-Snapshots
  - Teilweise Data Race Detection
    - Write/Write-Konflikte unter Tasks erkannt

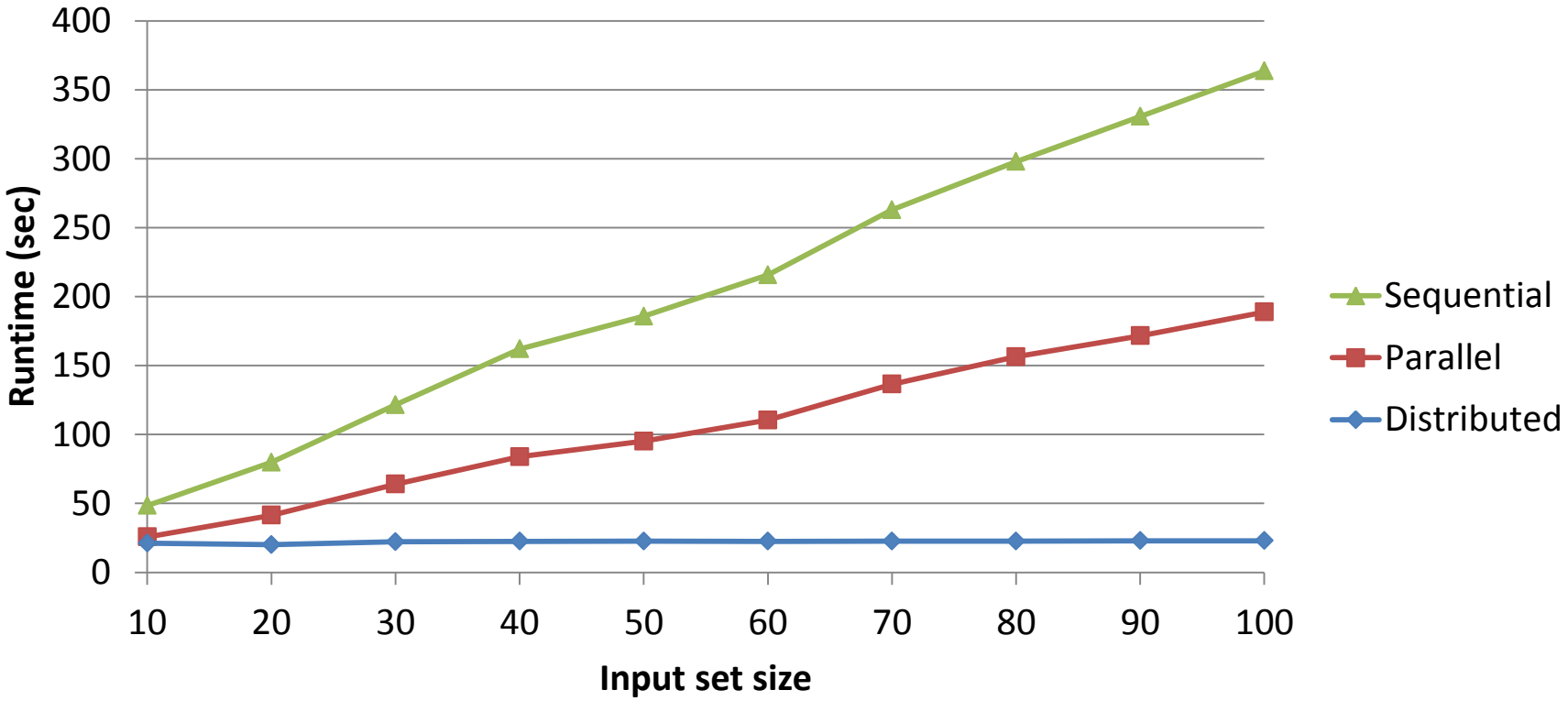
Stand der Implementierung

- Für Distributed Tasks noch nicht unterstützt
  - Vererbung, Interfaces, Delegates
  - Struct und Nullable Typen
  - Multidimensional Arrays
    - Geschachtelte Arrays sind unterstützt
  - Ref und Out Parameter
  - Exceptions
  - Task Canceling & Monitoring

  => Work in Progress

Performance-Skalierung

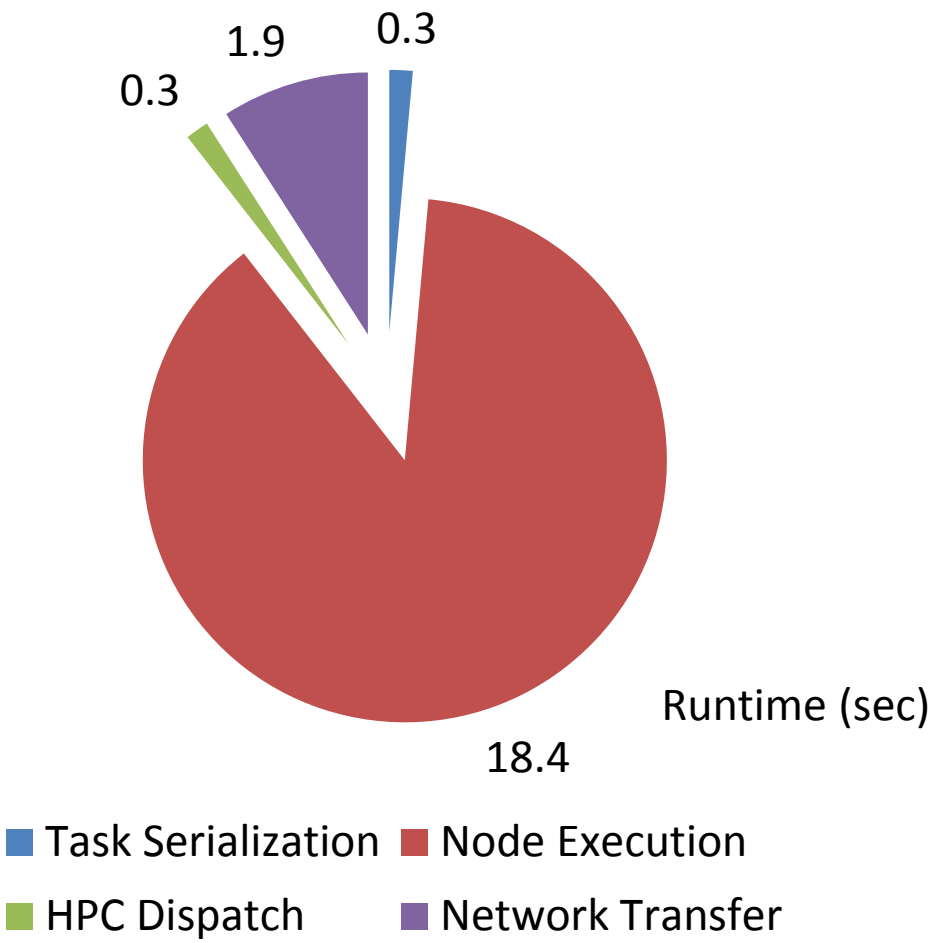
Primfaktorzerlegung



Factorize a set of predefined numbers; Minimum of 3 measurements;
Client Intel 2 Core, 2.9 GHz; Service Intel 2 Core, 2.9 GHz; 64 Bit, with Compiler Optimization
Cluster MS HPC 2012, 32 Nodes Intel Xeon 12 Core 2.6GHz; 100MBit/s network, 1ms delay

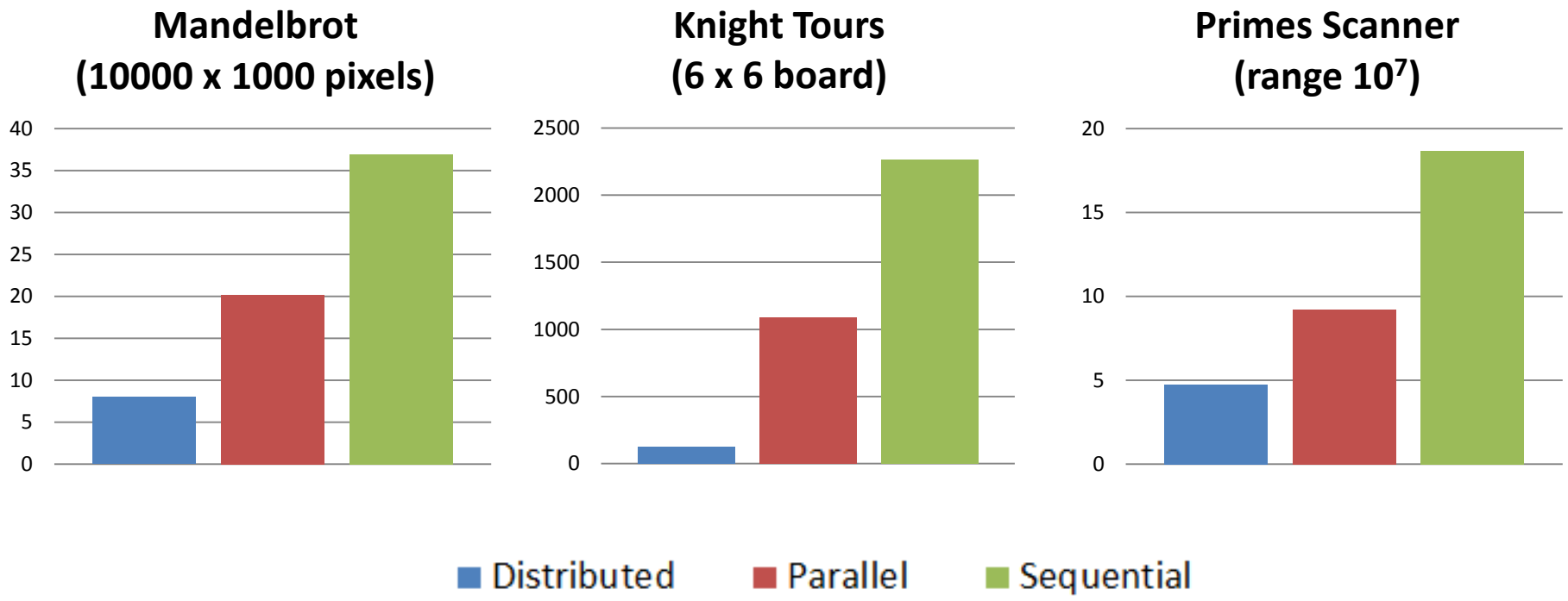
Performance Kostenanteile

Primfaktorzerlegung (10 Zahlen)



Performance-Vergleiche

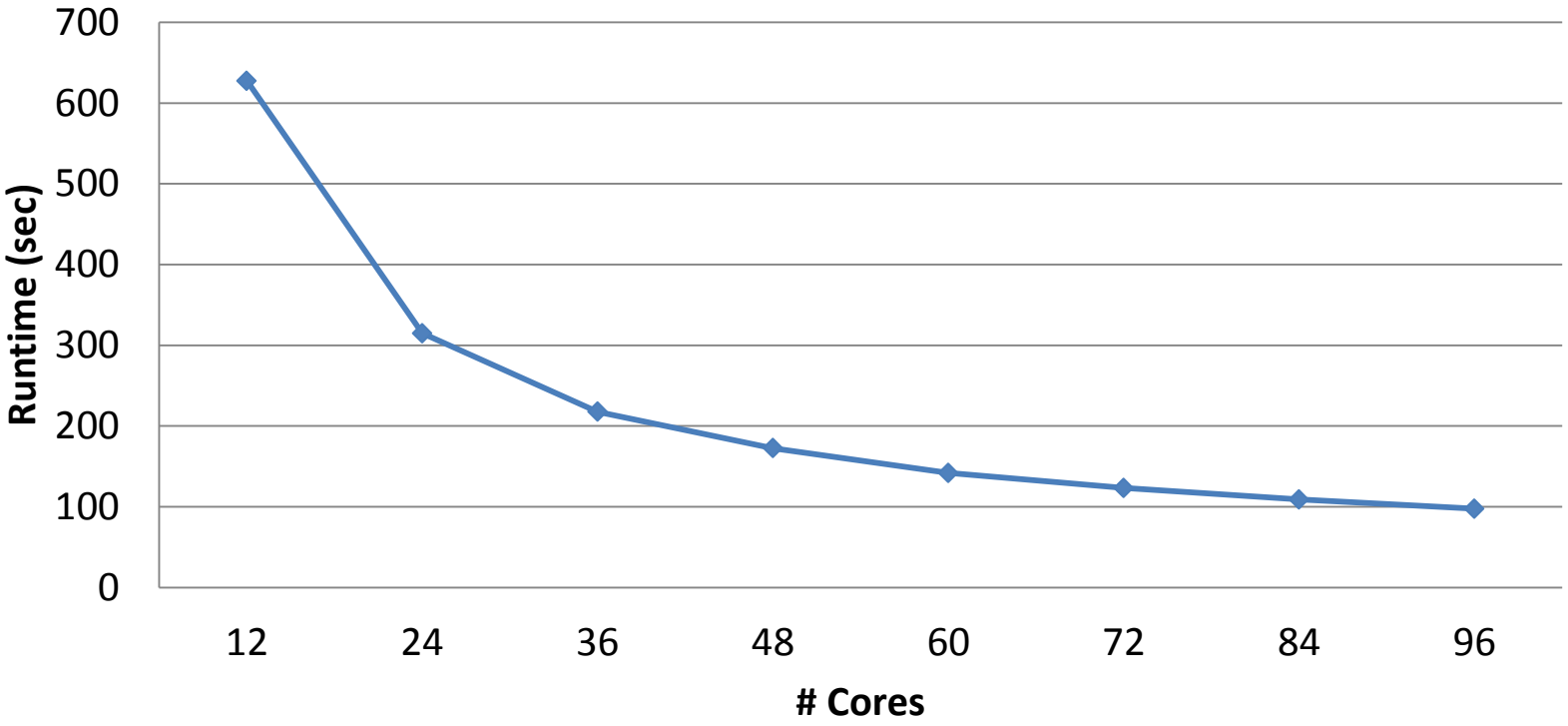
- Drei weitere Beispiele (Laufzeiten in Sekunden)



Minimum of 3 measurements; Client Intel 2 Core, 2.9 GHz; Service Intel 2 Core, 2.9 GHz; 64 Bit, with Compiler Optimization
Cluster MS HPC 2012, 32 Nodes Intel Xeon 12 Core 2.6GHz; 100MBit/s network, 1ms delay

Parallele Beschleunigung

- Laufzeit abhängig zur Anzahl Cluster Nodes/Cores



Factorization of 100 predefined input numbers
Client Intel 2 Core, 2.9 GHz; Service Intel 2 Core, 2.9 GHz; 64 Bit, with Compiler Optimization
Cluster MS HPC 2012, 32 Nodes Intel Xeon 12 Core 2.6GHz; 100MBit/s network, 1ms delay

Performance Diskussion

- Speedup
  - Hohe Parallelität durch viele leistungsfähige Cores
- Overheads
  - Übertragung zwischen Client und Backend
    - Durchsatz (Datenmenge) und Latenz (Netzdistanz)
  - Task Serialisierung/Deserialisierung
  - Dispatchen der Cluster-Jobs
- Parallelisierung muss Overheads kompensieren
  - Viele Tasks
  - Rechenintensive Tasks
  - Tasks mit wenig Datentransfer

Schlussfolgerungen

- Nahtlose verteilte Task-Parallelisierung in .NET
  - Prinzipiell gleiches Programmiermodel wie lokale Tasks
  - Illusion des Shared Memory Models trotz Verteilung
  - Keine explizite Entwicklung von Remote Code
  - Keine explizite Übertragung oder Kommunikation
  - Write/Write Race Detection als zusätzliche Sicherung

Related Work

- Microsoft Dryad LINQ
  - Verteilte Paralleliserung von .NET LINQ Ausdrücken
  - Keine Seiteneffekte / Heap Updates erlaubt (ohne Checks)
- Actors, MPI
  - Kommunizierende Agenten/Prozesse/Threads
  - Explizite Kommunikation, kein Shared Memory
- Grid Computing
  - Runtime über verschiedene Rechner
  - Weniger nahtlos: spezielle Grid Heap/Threads, oft explizit serialisierbare Objekte, explizites Remoting usw.

Ausblick & Ideen

- Parallelization-as-a-Service
  - Virtualisierung der reinen Prozessorressourcen
  - On-Demand oder Pay-to-Go Parallelization
- Technische Erweiterungen
  - Language Feature Set ausbauen
  - Model-Erweiterungen (Cancel/First Fetch, Interaktionen)

Danke für Ihr Interesse

- Projektwebseite
  - http://concurrency.ch/Projects/TaskParallelism
- Kontakt
  - **Prof. Dr. Luc Bläser**
    HSR Hochschule für Technik Rapperswil
    IFS Institut für Software
    Rapperswil, Schweiz
  - lblaeser@hsr.ch
  - http://ifs.hsr.ch
    http://concurrency.ch