Language and Runtime Innovations for Local and Distributed Parallelization

Luc Bläser
HSR Hochschule für Technik Rapperswil
lblaeser@hsr.ch

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

Modeling and Simulation Seminar
Chair of Sociology , ETH Zurich, 8 Apr 2013

Personal Research Overview

- Structured concurrency with Composita
  - A new programming language and runtime system with application to agent simulation
- Seamless distributed task parallelization
  - A runtime system extension for .NET for seamless distribution of parallel code onto HPC clusters

Structured Concurrency with Composita

Overview

Composita

- A new programming language
  - Hierarchical und well-structured components
  - Inherent and safe concurrency
- Traffic simulation as application study
  - Simulation is a predestined case for new languages
    - SIMULA: Birth of object-orientation motivated by simulations
  - Goals
    - Natural simulation modeling
    - High expressiveness
    - Reasonable performance

Motivation

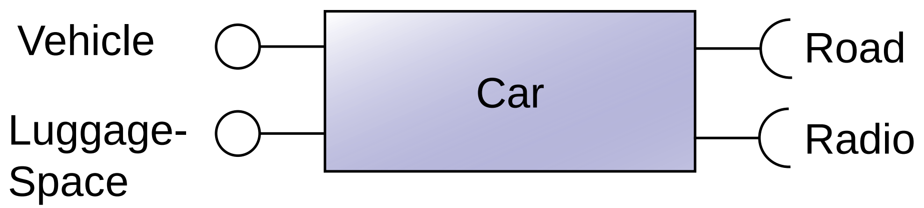
Problems of mainstream object-oriented languages

- References
  - Flat object structures without explicit hierarchies
  - Intended encapsulation is not guaranteed
- Inheritance
  - Forced combination of polymorphism and reuse
  - Limited single inheritance or multi-inheritance conflicts
- Concurrency
  - Unnecessarily blocking interactions via method calls
  - Threads operate on passive objects without proper control

Components in Composita

- General abstraction unit at runtime
  - Subjects (e.g. “person”), active objects (e.g. “car”), passive objects (e.g. “road”), abstract notions (e.g. “route”)
- Strict encapsulation
  - External dependencies only allowed via explicit interfaces
- Component can offer and require interfaces
  - Offered interfaces represent own external facets of a component
  - Required interfaces are to be provided by other components
- Multi-instantiation from a component template

COMPONENT **Car**
  OFFERS **Vehicle**, **LuggageSpace**
  REQUIRES **Road**, **Radio**
  (* implementation *)
END Car



Hierarchical Composition

- A component can contain an arbitrary number of subcomponents

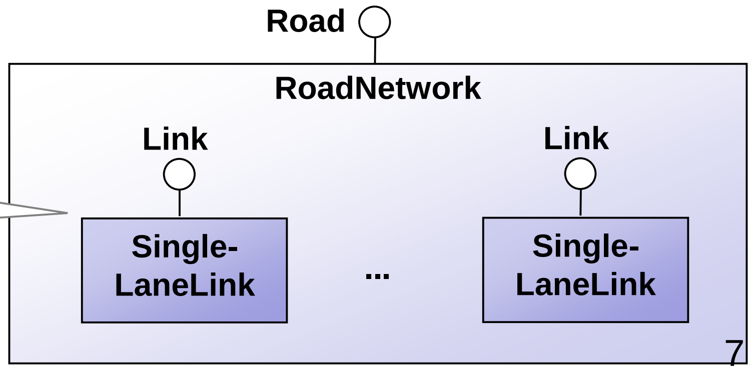
Dynamic collection of subcomponents

COMPONENT RoadNetwork OFFERS Road;
VARIABLE **link[id: INTEGER]: ANY(Link)**;
BEGIN
  FOREACH *link* id *in simulation file* DO
    NEW(link[id], SingleLaneLink)
  END
END RoadNetwork;

Arbitrary type with postulated interfaces

Component template

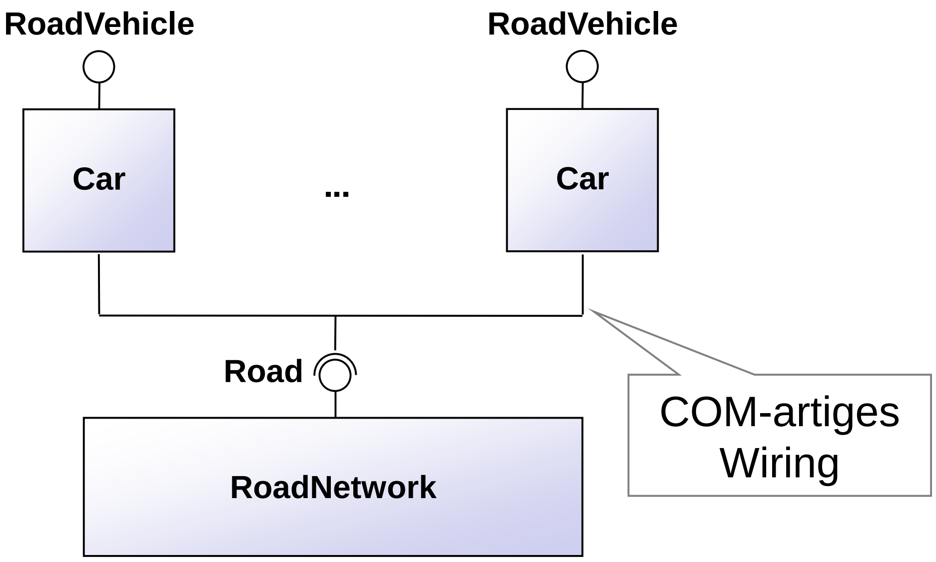
Encapsulated subcomponents



Interface Connections

- Each required interface can be connected to an offered interface with the same name

```
COMPONENT TrafficSimulation;
VARIABLE
  car[id: INTEGER]: Car;
  road: RoadNetwork
BEGIN
  NEW(road);
  FOREACH car id in simulation file DO
    NEW(car[id]);
    CONNECT(Road(car[id]), road)
  END
END TrafficSimulation;
```

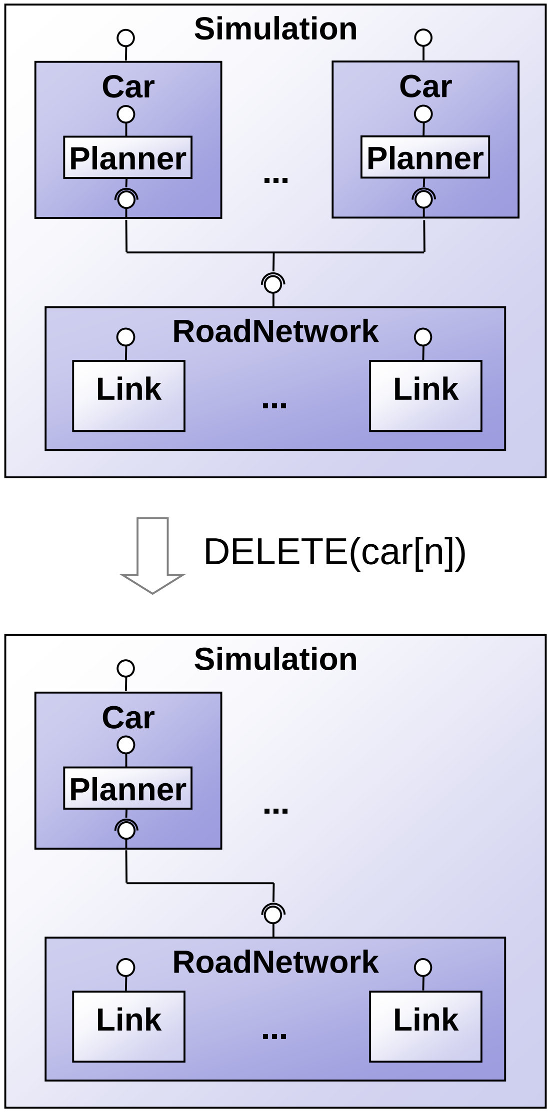


No ordinary pointers

- Connections are exclusively set by the surrounding component
- Outgoing and incoming interface points explicitly defined per component

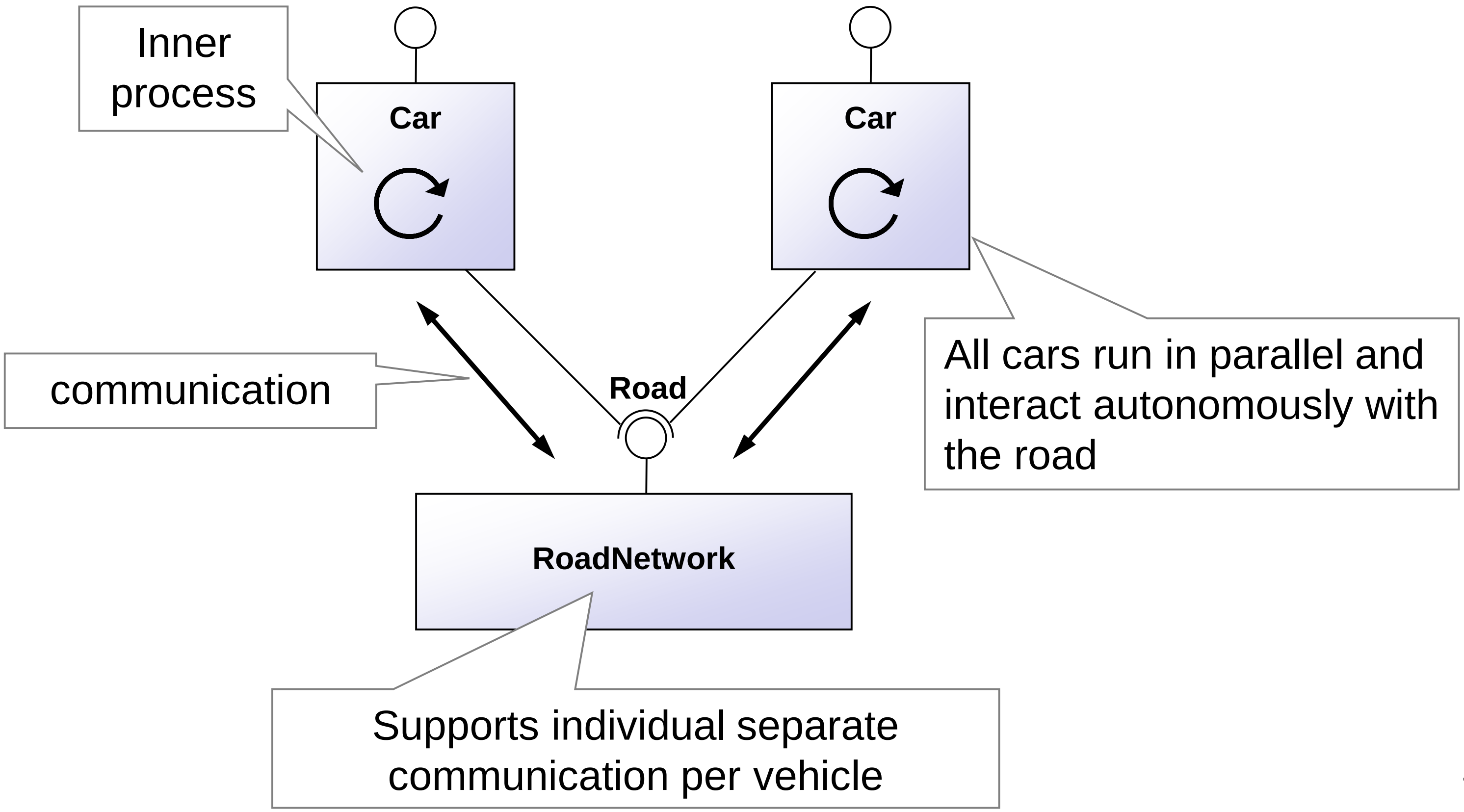
Memory Management

- Hierarchy of component networks
  - Network structure is exclusively governed by surrounding component
- Hierarchical existence dependencies
  - Deletion of a component => Automatic deletion of subcomponents
  - Explicit deletion of a component => interfaces become disconnected in a controlled way
  - Memory safety without garbage collection (no low-level dangling pointers or memory leaks)



Concurrency and Interactions

- Each component runs its own inner light-weight processes
- Components interact only by way of communication over interfaces



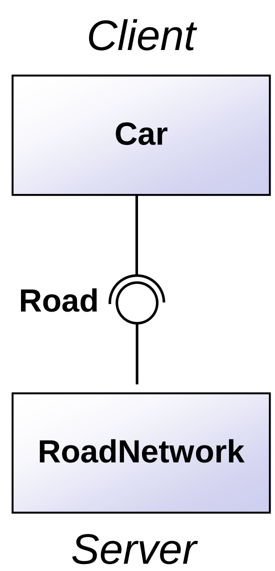
Communication

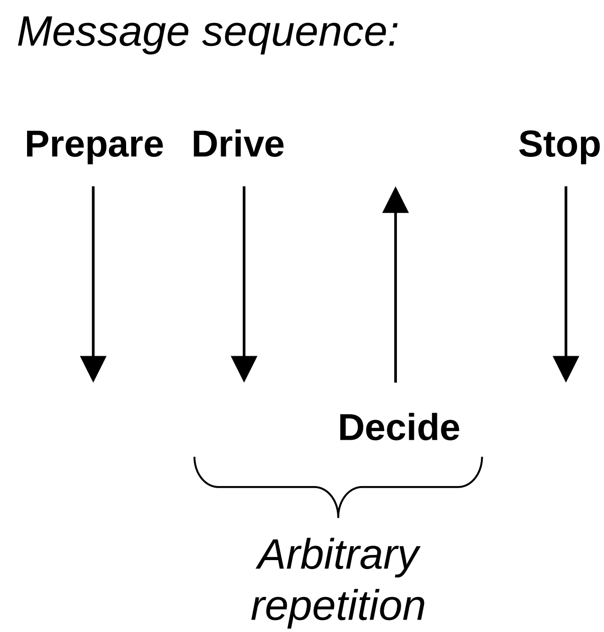
- Separate communication between each client and server
- Sending and receiving messages according to a formal protocol

```
INTERFACE Road;
  IN Start
  {
    IN Drive(linkId: INTEGER)
    OUT Decide(time: INTEGER)
  }
  IN Stop
END Road;
```

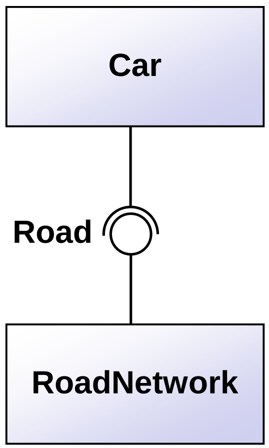
repetition

Communication protocol in EBNF





Component Implementations




```
COMPONENT Car (* … *) REQUIRES Road;
BEGIN
  Road!Start;
  WHILE target not reached DO
    Road!Drive(nextLinkId); Road?Decide(time)
  END;
  Road!Stop
END Car;
```

Send message

Receive message

```
COMPONENT RoadNetwork OFFERS Road;
  IMPLEMENTATION Road;
  BEGIN
    ?Start;
    WHILE ?Drive DO
      ?Drive(linkId); (* drive *) !Decide(now)
    END;
    ?Stop
  END Road;
END RoadNetwork;
```

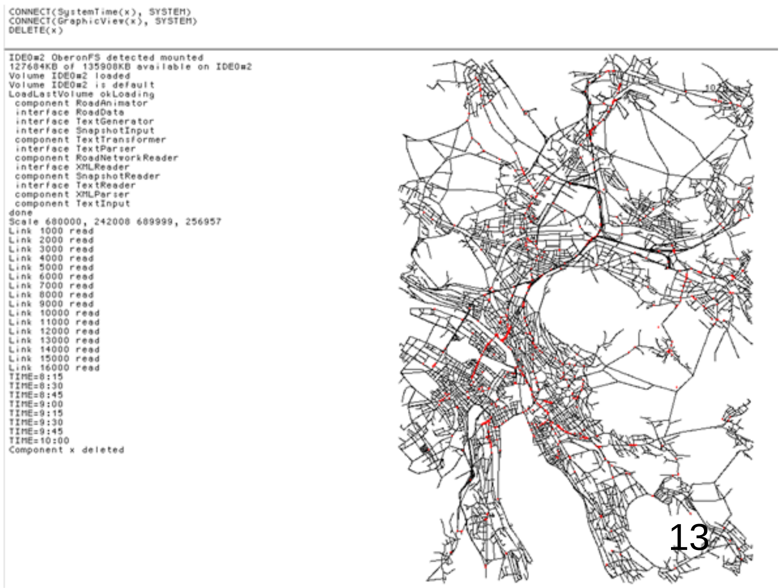
Separate service process per client

Receival test

Runtime System

A small operating system for scalable efficient concurrency

- Light-weight processes
  - Dynamic micro stacks
  - Enables huge amount of processes
- Fast context switches
  - Direct synchronous switches
  - Preemption with code instrumentation
- Inbuilt synchronization
  - Protocol-based communication
  - System-managed monitors
- Efficient memory management
  - Hierarchical memory management
  - No virtual memory management



Scaling and Performance

- Maximum number of threads / light weight-processes

Component OS	Windows .NET	Windows JVM	Active Oberon
5,010,000	1,890	10,000	15,700

4GB main memory

- Execution performance

Program (sec)	Component OS	C#	Java	Oberon AOS
ProducerCons.	**16**	19	130	60
Eratosthenes	**1.8**	6.8	4.6	5.8
TokenRing	**2.1**	22	22	18

6 CPUs Intel Xeon 700MHz

Traffic Simulation Study (with TU Berlin)

Developed in the new language

- Self-active cars
  - All cars drive autonomously and concurrently
  - No explicit program loop, centrally controlling the car movements
  - No explicit parking and waiting queues
- Virtual time
  - Virtual time corresponds to the time in the simulated world
  - All cars run with a synchronous virtual time
- Individual planning and learning
  - Drivers plan their journey, route and departure time individually
  - Each driver thereby considers their own experience of previous journeys (traffic delays)

A Simplified Road Link

```
COMPONENT SingleLaneLink OFFERS Link;
  VARIABLE occupied[cell: INTEGER]: BOOLEAN;

  IMPLEMENTATION Link;
  VARIABLE cell: INTEGER;
  BEGIN {EXCLUSIVE}
    AWAIT(~occupied[0]); occupied[0] := TRUE;
    !Entered; PASSIVATE(1); cell := 0;
    WHILE cell < length DO
      AWAIT(~occupied[cell + 1]);
      occupied[cell + 1] := TRUE;
      occupied[cell] := FALSE;
      INC(cell);
      PASSIVATE(1)
    END;
    !EndReached;
    occupied[exit] := FALSE
  END Link;
END SingleLaneLink;
```

Cellular automaton

Autonomous driving process per car on link

Monitor lock

Wait for next free cell

Wait a virtual second

AWAIT & PASSIVATE temporarily release monitor lock

Runtime Performance

Scales with amount of concurrent cars & traffic jam

Scales with length of road & time slices

Zurich traffic simulation (in minutes)	Composita (concurrent, virtual clock)	C++ (sequential time-sliced)	C# (multi-threaded, virtual clock)
1,000 cars	0.04	140	**33**
10,000 cars	0.6	140	**out of memory**
100,000 cars	13	190	**out of memory**
260,000 cars	**76**	**210**	**out of memory**

Different simulation model!

Analogous model but threads not designed for large scale

6 CPUs Intel Xeon 700MHz

Conclusions

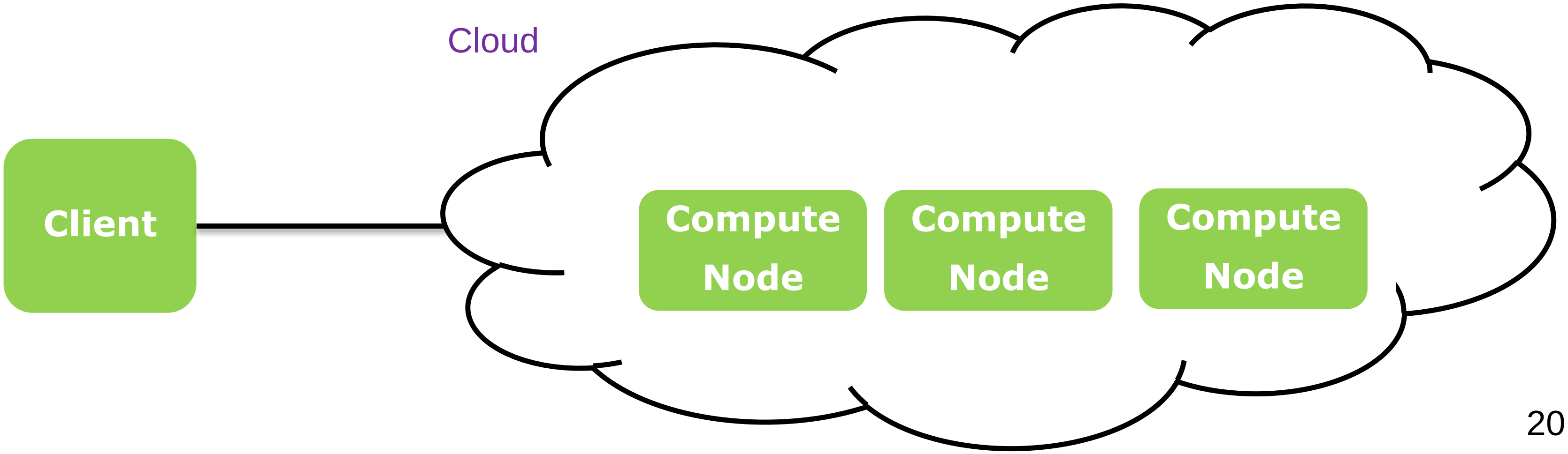
- Natural simulation description
  - Autonomous driving behavior per car
  - Cars run in parallel
  - Driving based on a virtual clock
  - Individual planning an learning
  - Abandoned artifacts
    - Explicit park and wait queues
    - Global program loop for discrete event queue / time-slices
    - Centralized event recording and planning
- Flexible programming
  - Components could be programmed a simple and compact manner
  - New structures able to seamlessly replace ordinary references/pointers
- Good execution performance
  - In our case: Faster than analogous multi-threading and sequential time-sliced simulation

Seamless Distributed Task Parallelization

Overview
(ongoing project)

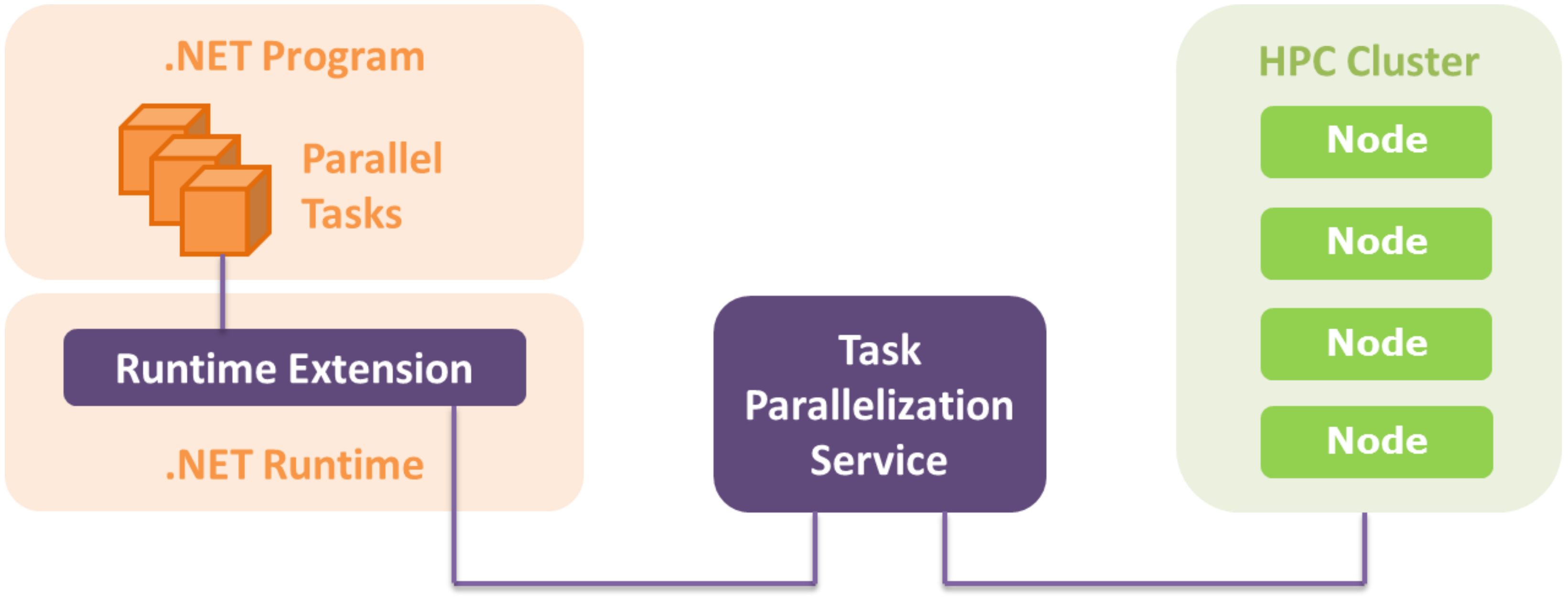
Goal: Parallelization in the Cloud

- Embed remote computing power locally
  - Massive parallelization in the cloud
  - E.g. on cluster with many multi-core nodes
- As seamless and simple as possible
  - Same programming model as on local cores
  - No explicit transmission or remote code needed



Cloud Task Parallelization in .NET

- Program parallel tasks in.NET
- Automatic deployment and execution in cloud



Classical .NET Task Parallelization

Factorize multiple numbers in parallel

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

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

Start task

Task delegate (lambda)

Await task completion

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

New Cloud Task Parallelization

Analogous to classical tasks

Denote service

Create task

Start multiple tasks in a bunch

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

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

distribution.Start(taskList);

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

Data Parallelism

Classical .NET Parallelization

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

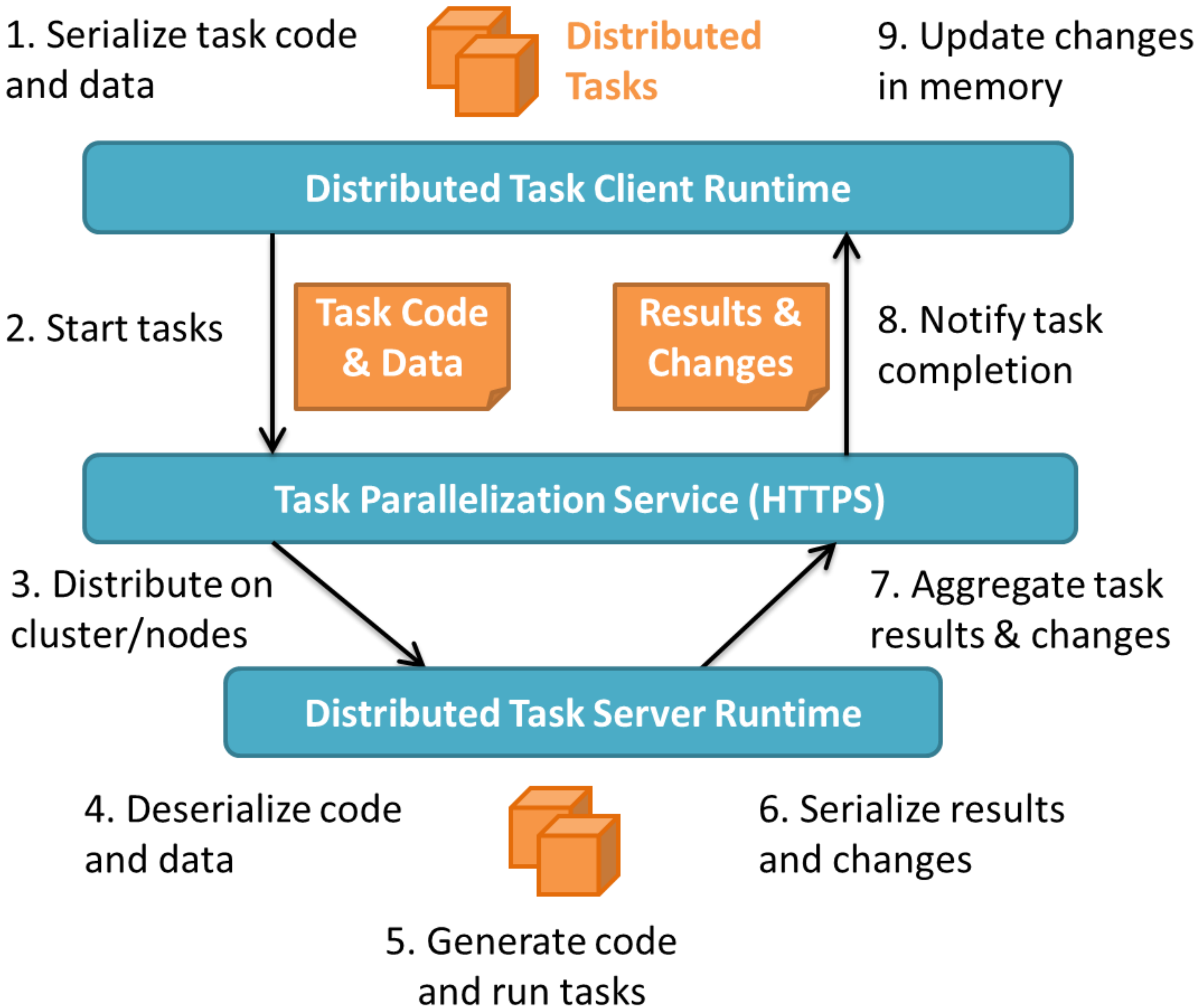
New Cloud Task Parallelization

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

Distributed Tasks

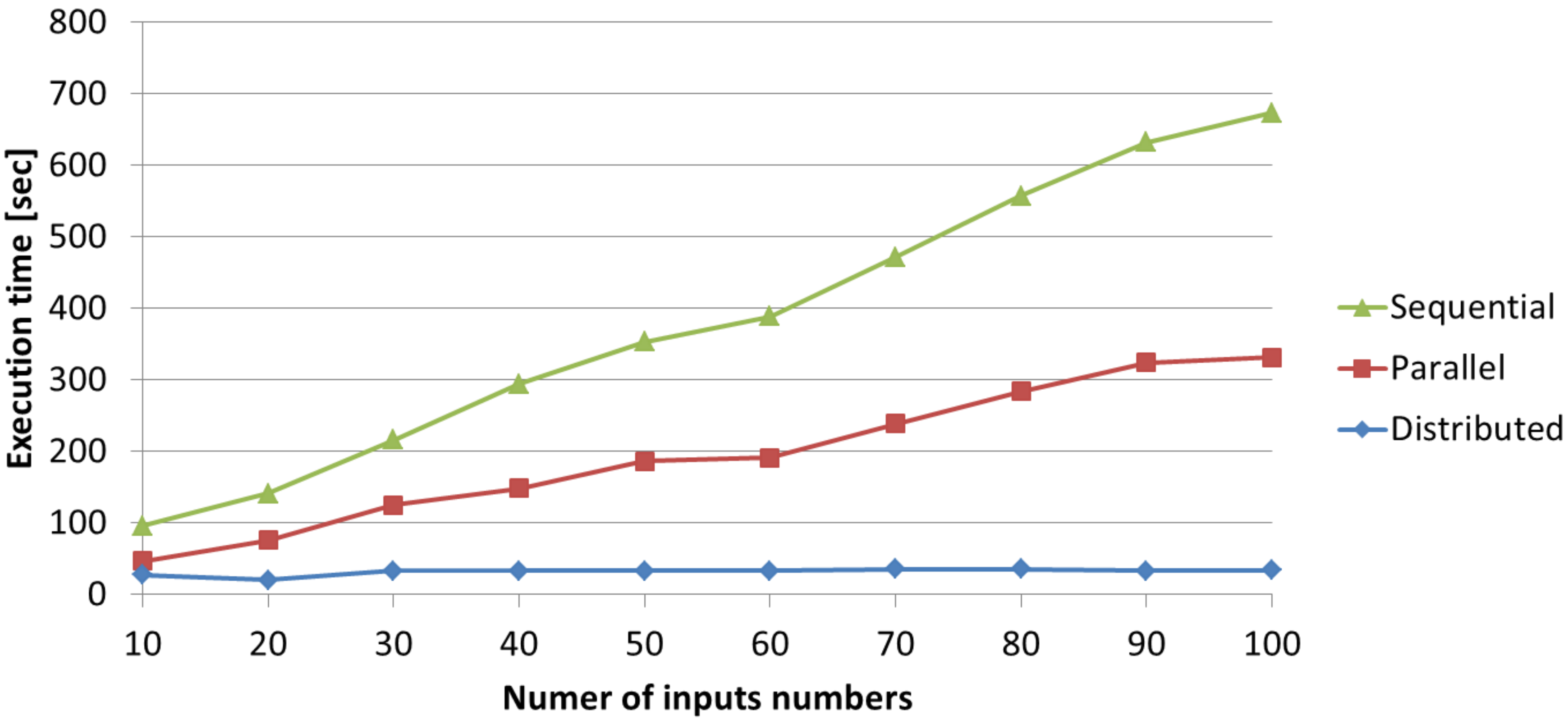
- Very similar to classical local tasks
  - Import of a library: no compilation step needed
- Bundled task start
  - Minimization of network roundtrips
- Task as a .NET delegate/lambda
  - General programming model
- Tasks need to be independent / isolated
  - Accesses on disjoint fields/array elements - except read-only accesses
  - Write/write conflicts detected by runtime system

Runtime System



Performance Scaling

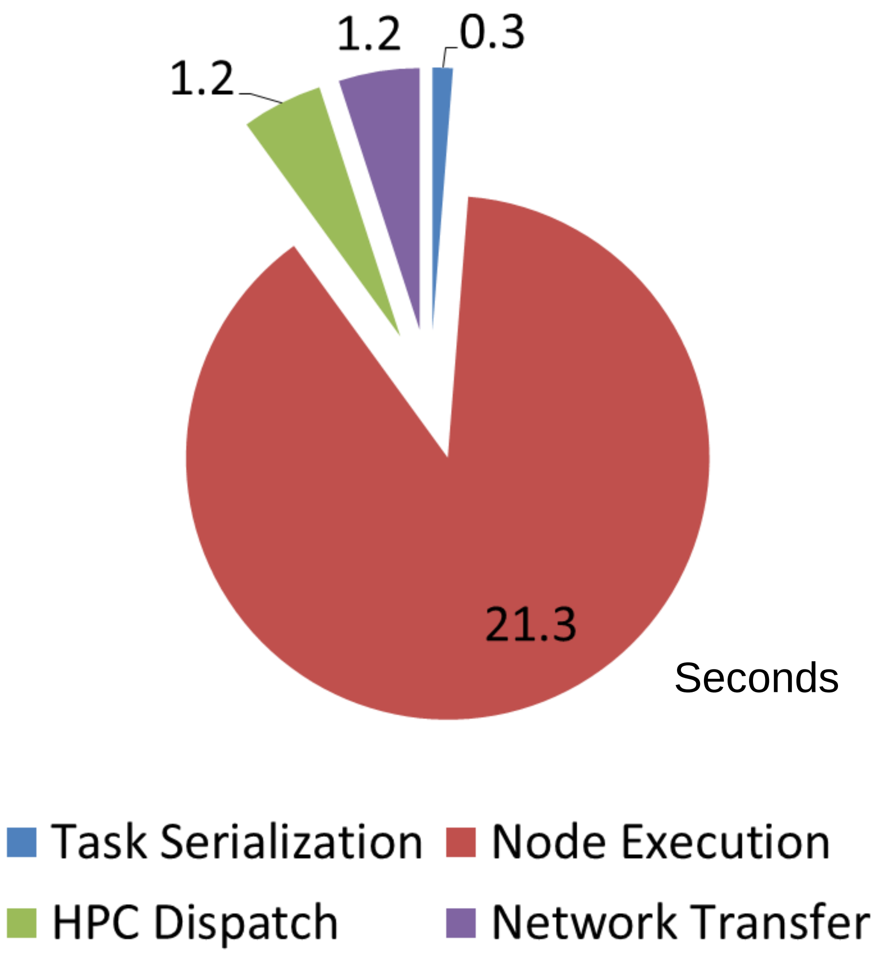
Prime factorization



Client and service: Intel 2 Core, 2.9 GHz
MS HPC 2008, 32 Nodes Intel Xeon 12 Core 2.6GHz
Network delay: 1ms, throughput 100 Mbit/sec

Performance Cost Amounts

Prime factorization (100 numbers)



Performance Discussion

- Speedup
  - High speedup by many powerful cores
- Overheads
  - Communication between client and backend
    - Throughput (data amount) und latency (network distance)
  - Task serialization / deserialization
  - Dispatching of HPC cluster jobs
- Parallelization needs to compensate overheads
  - Large amount of tasks
  - Compute-intense tasks
  - Tasks with little data traffic

Conclusions

- Seamless distributed task parallelization in .NET
  - Programming model equivalent to local tasks
  - Illusion of shared memory model despite distribution
  - No explicit development of remote code
  - No explicit transmission or communication
  - Write/write conflict detection for additional safety

Many thanks for your interest!

Questions?