.NET Task Parallelization in the Cloud

Runtime Support for Seamless Distribution of Shared Memory Parallel Tasks

Luc Bläser

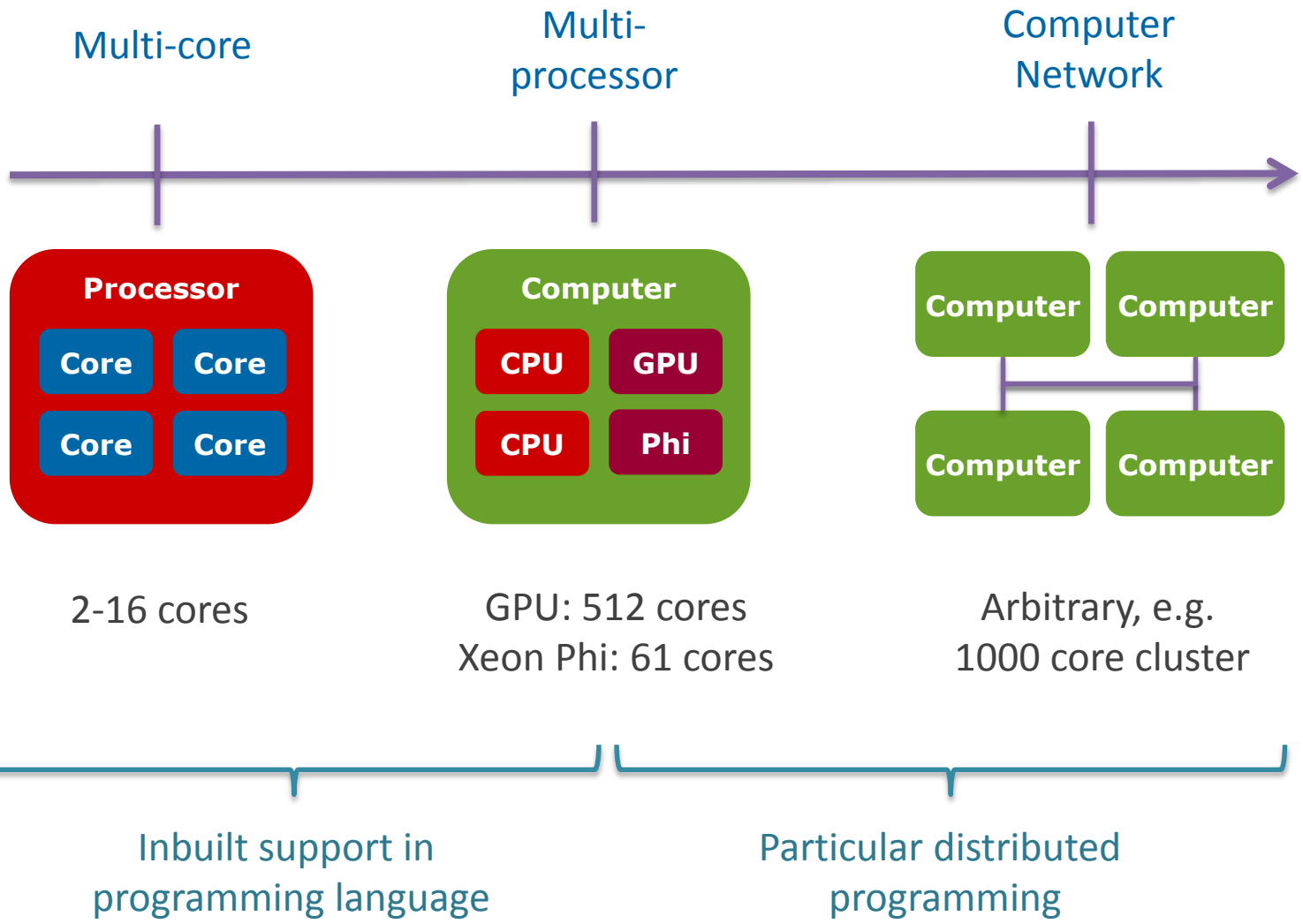
HSR University of Applied Sciences Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUT
FÜR
SOFTWARE

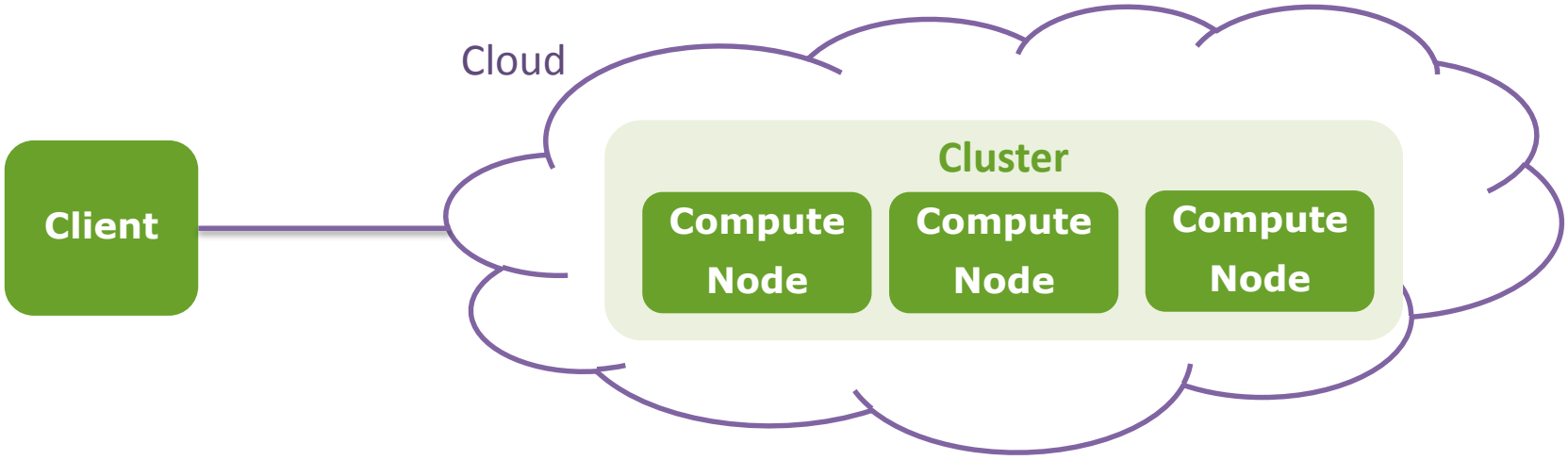
SFMA'14 @ Eurosys
13 April 2014

Levels of Parallelization



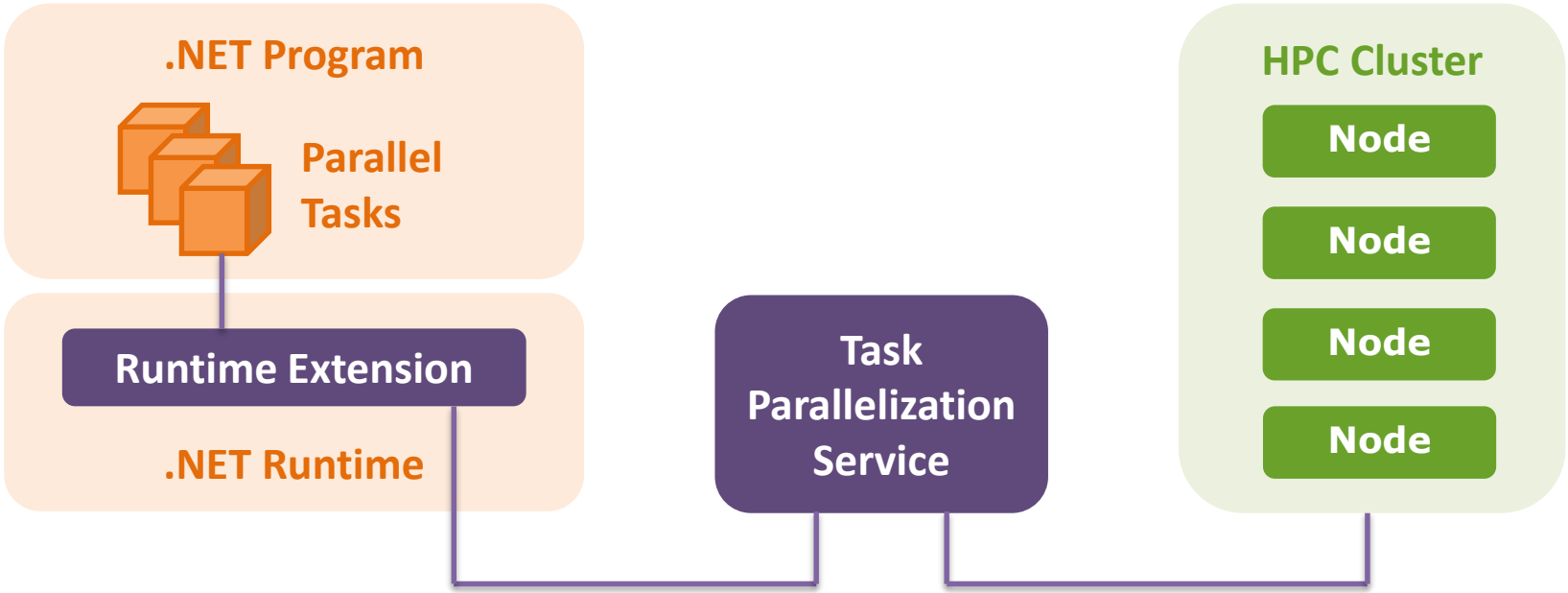
Seamless Distributed Task Parallelization

- Integrate remote processor power locally
  - Offer massive parallelization via a service
  - E.g. a many-core cluster behind the service
- **Easy-to-use and transparent for programmers**
  - Same programming model as for local cores
  - No explicit/visible separation of client/server code



.NET Task Parallelization in the Cloud

- Program parallel tasks in .NET (shared memory)
- Automatically send them to the cloud for execution
- Cloud side uses for example a MS HPC cluster



Overview

- Programming model
- Runtime system
- Experimental results
- Conclusions

Classical .NET Task Parallelization

Factorize multiple numbers

```
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);
}
```

Start TPL task

Task delegate (lambda)

Await task end

```
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 TPL

Specify service

```
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);
}
```

Create task

Start multiple tasks at once

Reference library: `HSR.CloudTaskParallelism.Client.Runtime`

Data Parallelization

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

- Nearly identical to TPL
  - Only import of a library: no compile step
- Bundled task starts
  - Minimizing network roundtrips
- Task as .NET delegate/lambda
  - Standard shared memory programming model
- Tasks must be independent
  - No shared mutable state

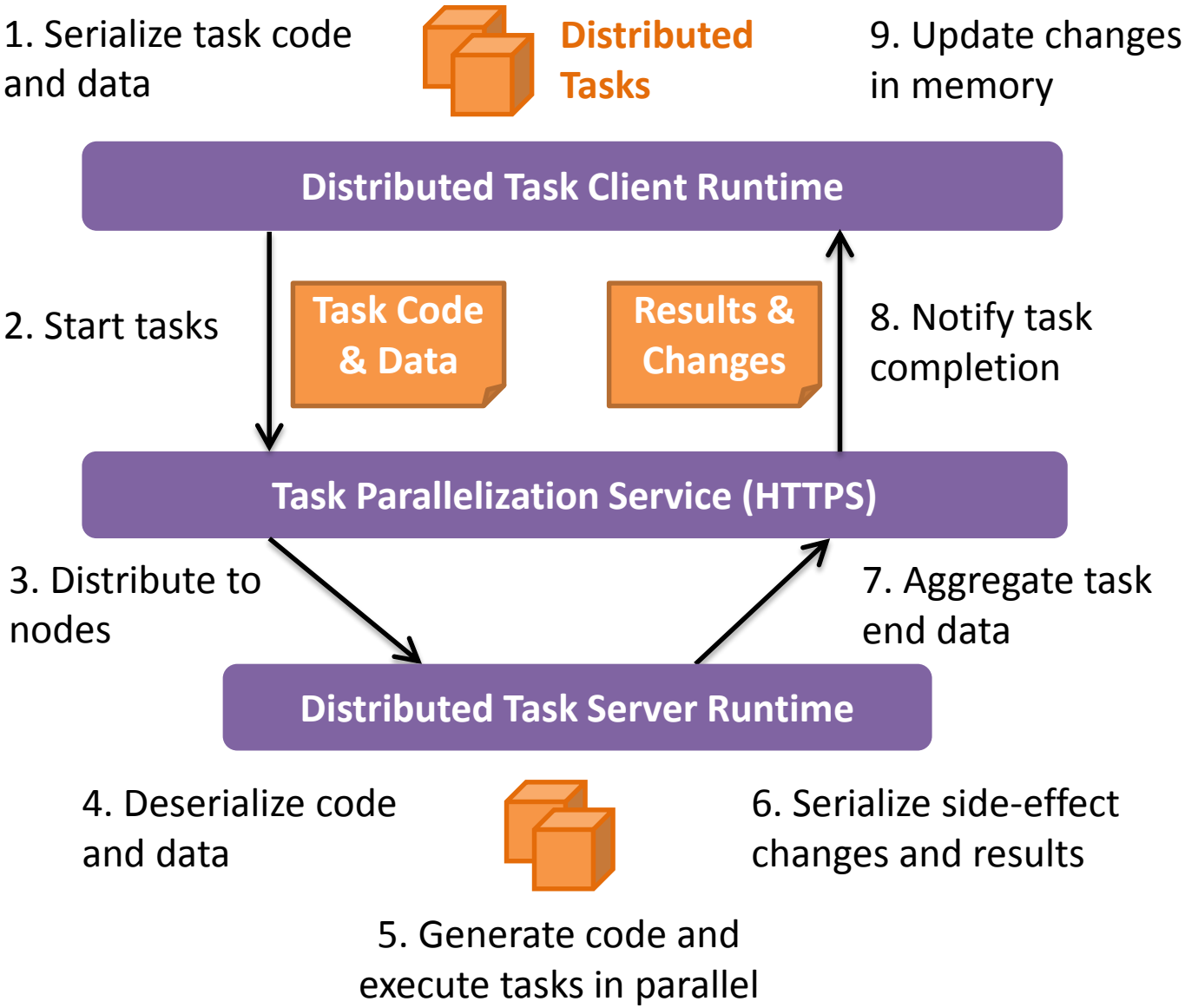
Task Independence

- Keeping it simple but efficient
  - Arbitrary distribution of tasks is feasible
  - Effortless parallel scalability
- How restrictive is it in practice?
  - Parallel decomposition is about minimizing synchronization – also for local tasks
  - Synchronization can often be avoided by different design, e.g. sequential post-phase for aggregating partial results
  - Programmer gains control over scalability, not leaving it to runtime heuristics/analysis

Distributed Task API

- Start of distributed tasks
  - `distribution.Start(taskSet)`
- Await task termination
  - `distribution.Await(taskSet)`
- Start tasks with wait barrier
  - `distribution.Invoke(taskSet)`
- `ParallelFor`, `ParallelForEach`

Runtime System



Task Serialization

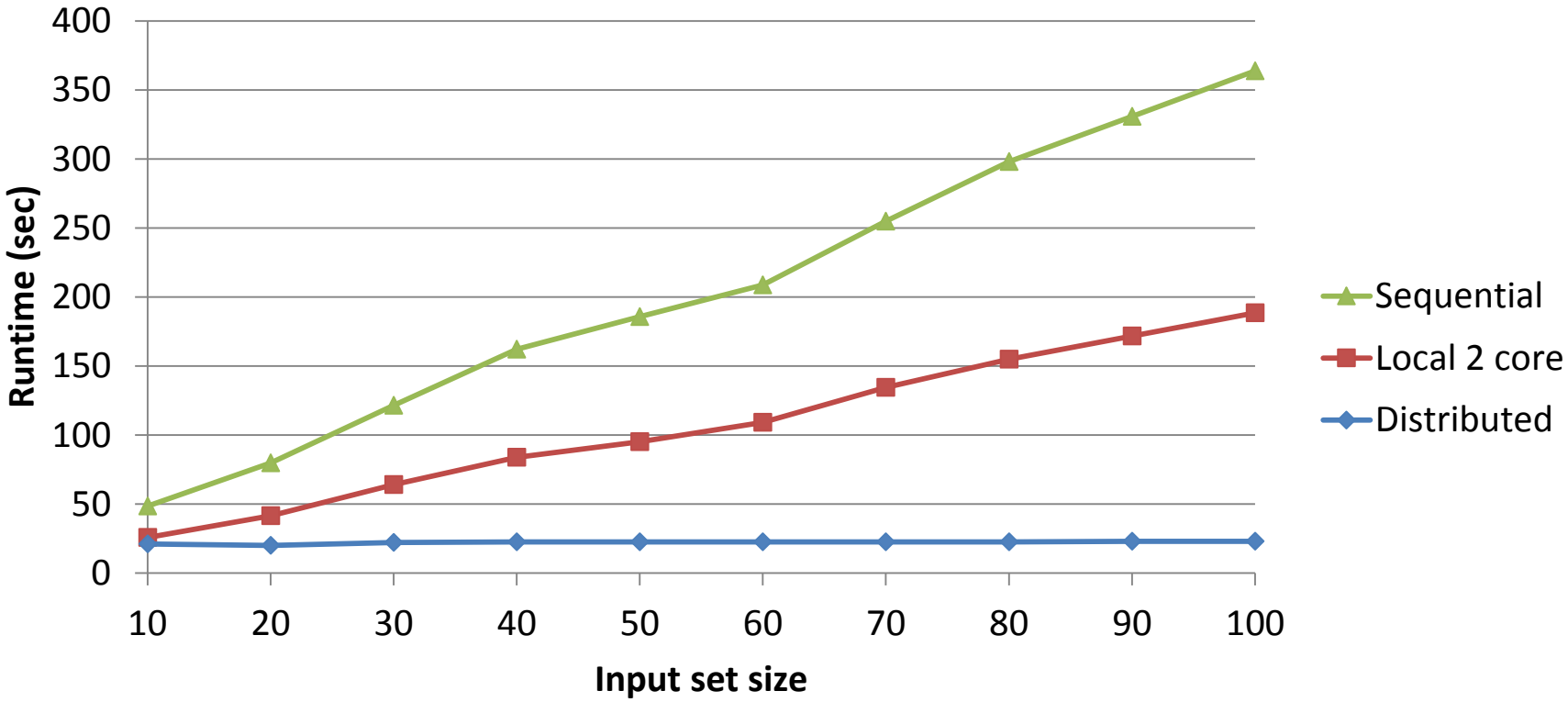
- Potentially executable task code
  - Conservative code analysis
    - Starting from task delegate
    - Directly and indirectly callable methods
    - Potentially used classes and fields
- Potentially accessed task data
  - Partial heap snapshot
    - Graph of reachable objects with accessible fields
    - Accessible static fields / constants
    - Consistency because of task independence

Task Updates/Results

- Delivered by the server on task completion
  - Task delegate result value
  - Changes in objects and static fields
    - Field updates
    - Array element updates
  - New allocated objects
- In-place updates at the client side
  - On the corresponding objects of the input snapshot
  - Partial data race detection
    - Write/write conflicts between distributed tasks

Performance Scaling

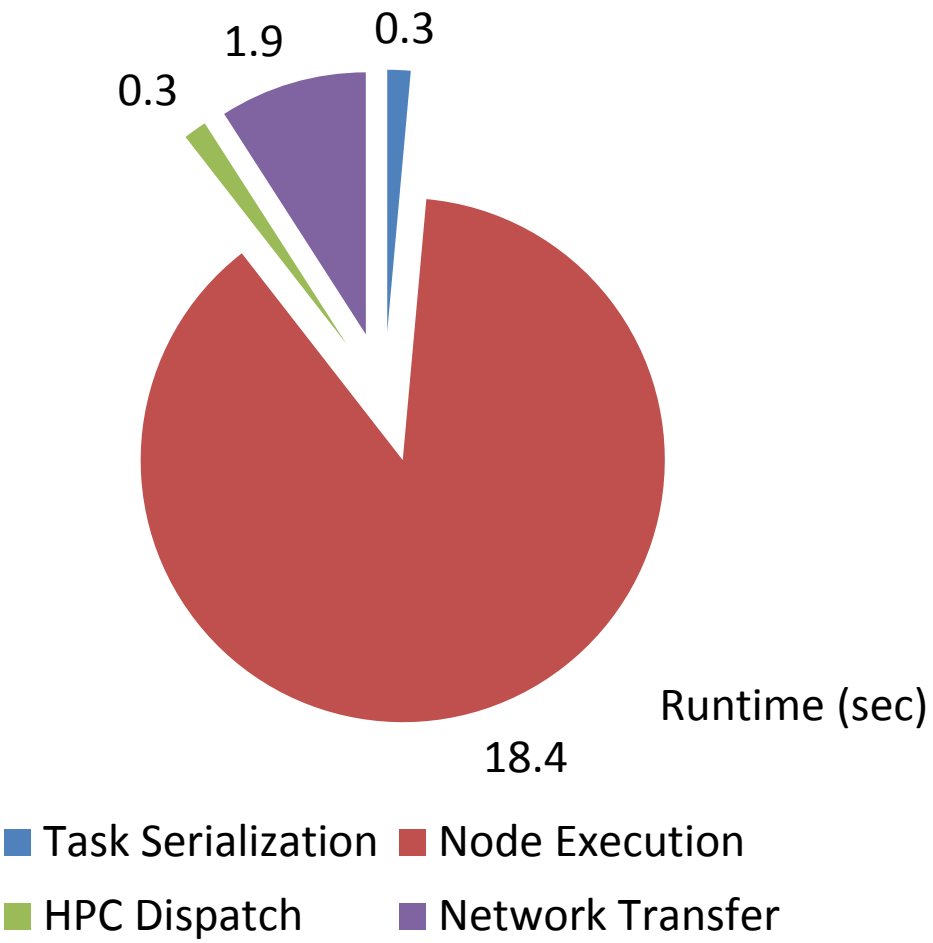
Prime factorization



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

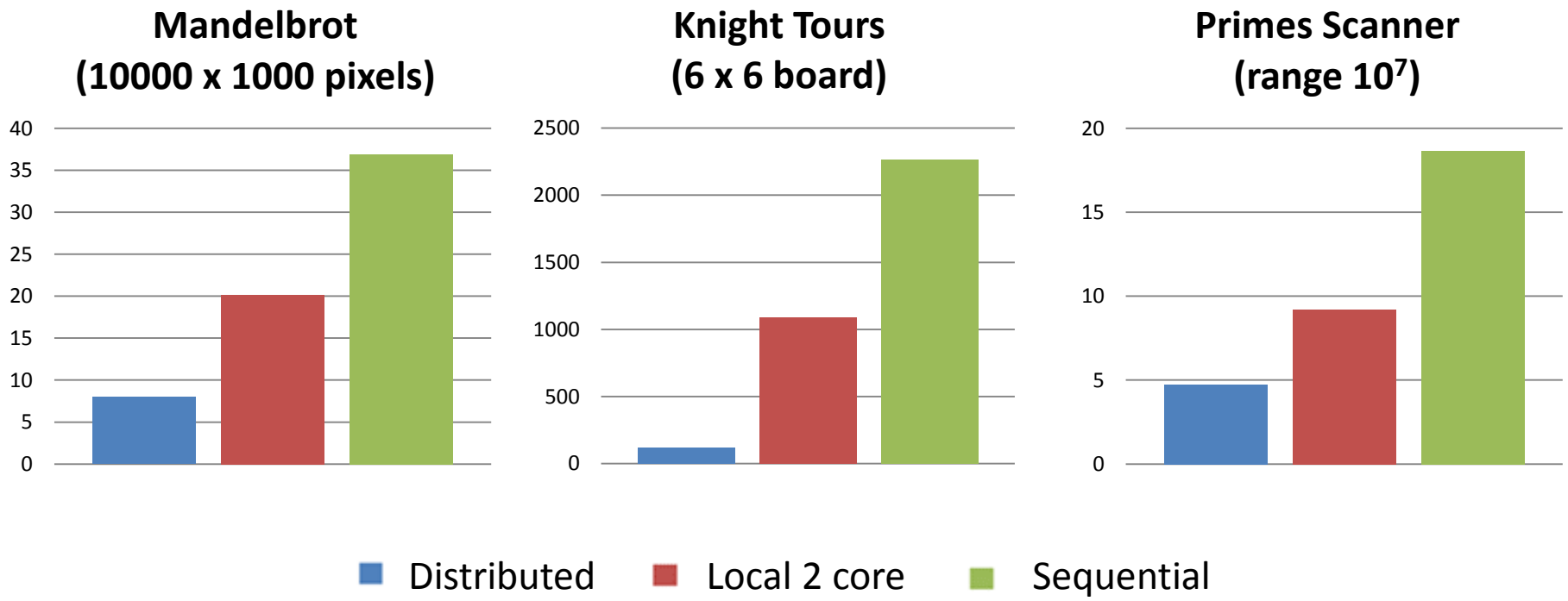
Performance Cost Breakdown

Prime factorization (10 numbers)



Performance Comparisons

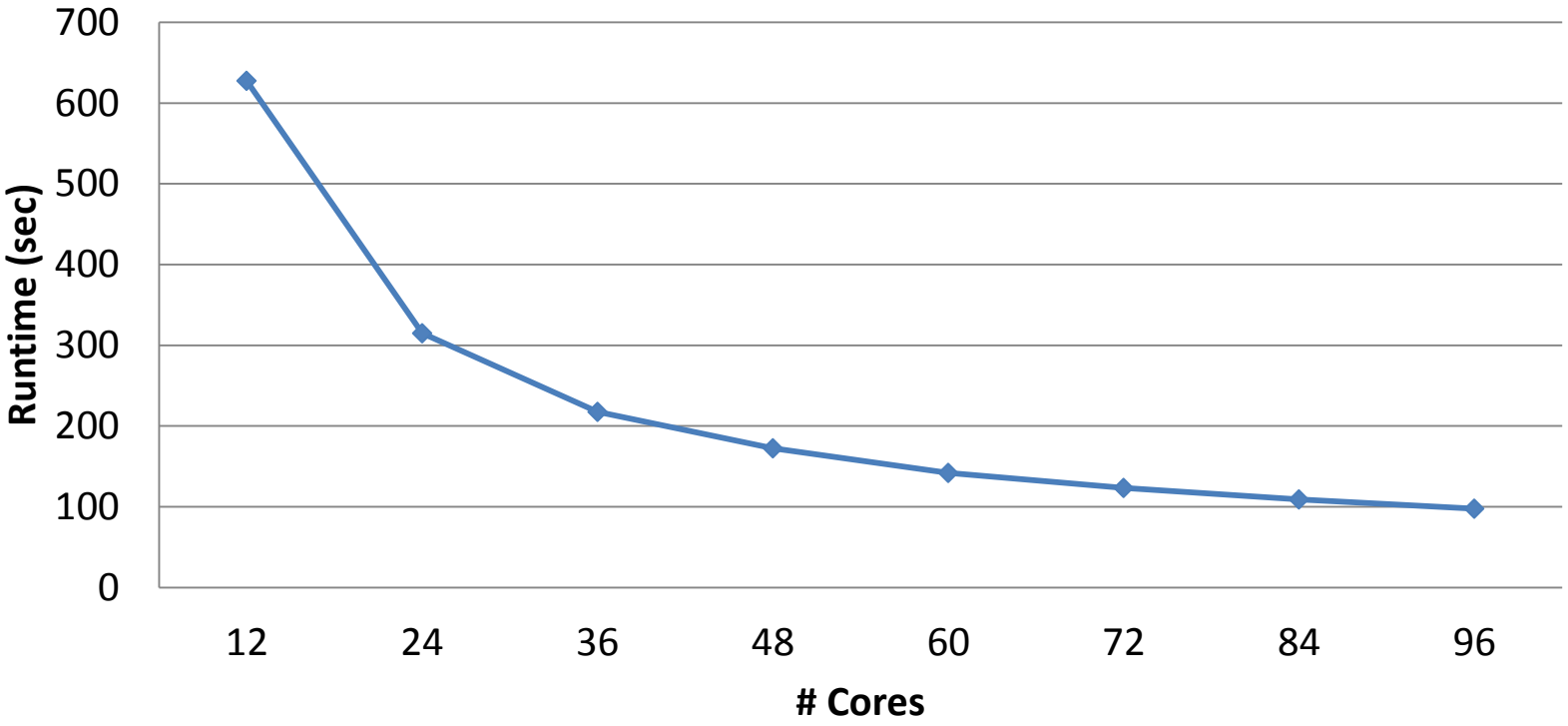
- Three more examples (runtimes in seconds)



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

Parallel Speedup

- Depending on number of used cores in cluster



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 Discussion

- Speedup
  - High parallelization by many general-purpose cores (CPUs)
- Overheads
  - Transmission between client and backend
    - Throughput (data amount) und latency (network delay)
  - Task serialization / deserialization
  - Dispatching of the HPC cluster job
- Parallelization needs to compensate overheads
  - Many Tasks
  - Compute-intense Tasks
  - Tasks with relatively small data amount
  - Depending on network / server settings

Conclusion

- Runtime for seamless distributed task parallelization
  - Principally same programming model as for local tasks
  - Illusion of shared memory models despite distribution
  - No explicit design of remote code
  - No explicit serialization (wrapping/marking/attributing code for distribution-awareness)
  - No explicit distribution or communication logic
  - Write/write race detection as extra safeguard

http://concurrency.ch/Projects/TaskParallelism