.NET Task Parallelization as A Service

A Runtime System for Automatic Shared Task Distribution

Luc Bläser

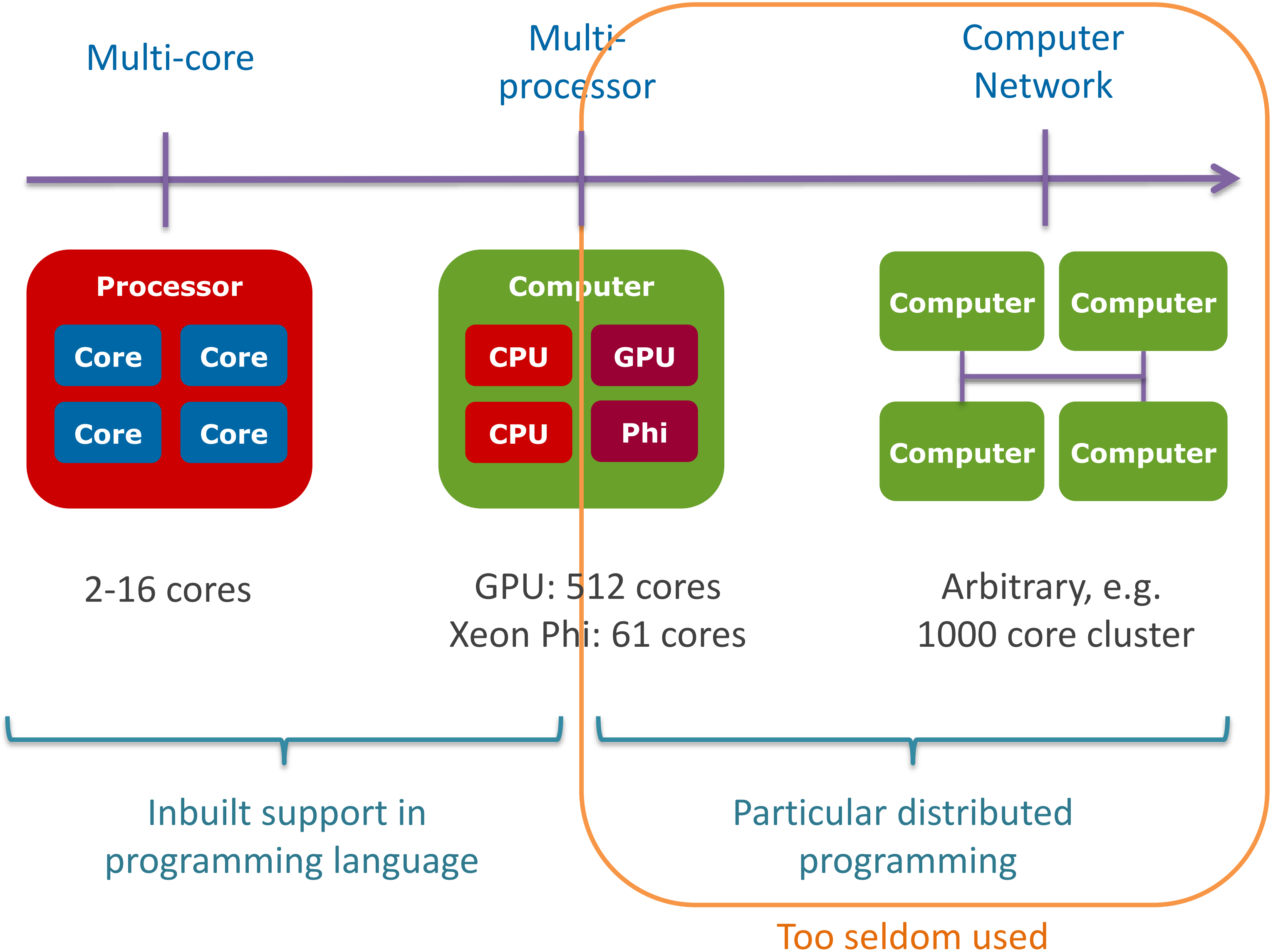
HSR University of Applied Sciences Rapperswil

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUT FÜR SOFTWARE

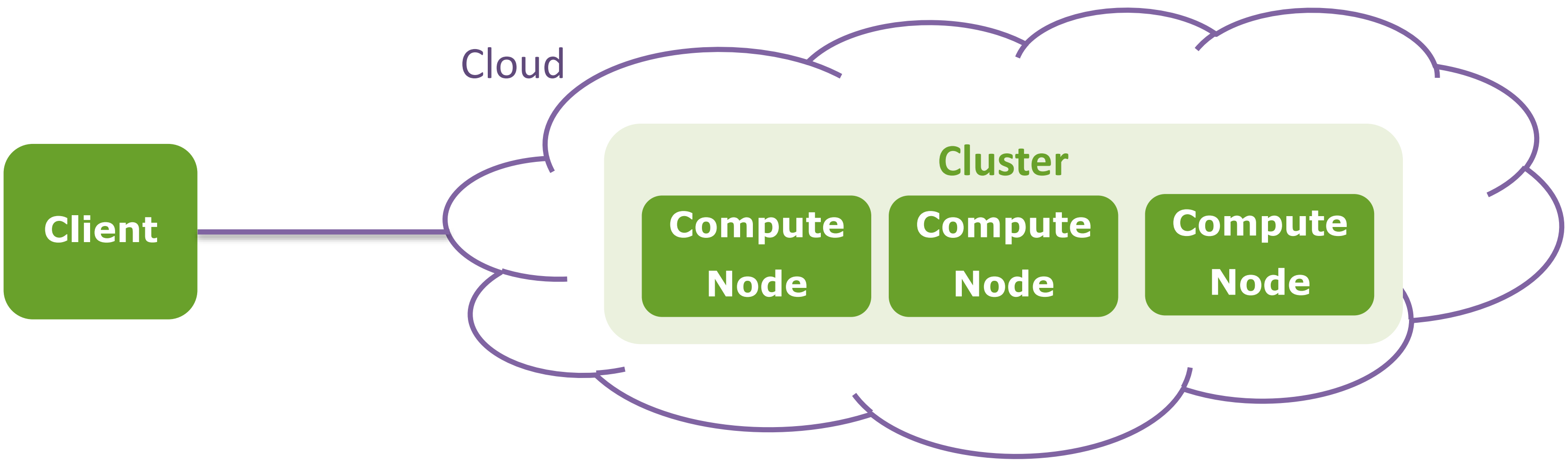
MULTIPROG2015 @ Hipeac
20 April 2015

Levels of Parallelization



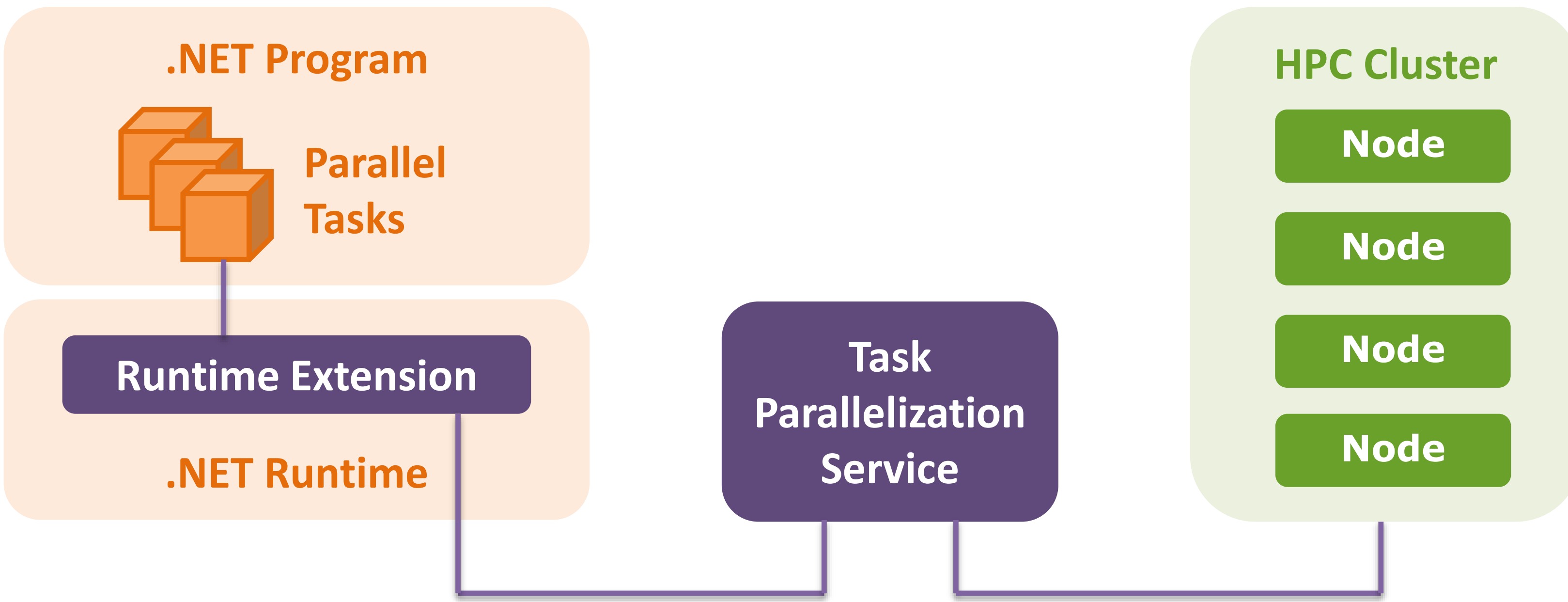
Task Parallelization as a Service

- 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 Shared Memory Task Distribution

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



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 Distributed Task Parallelization


```
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);
}
```

Data Parallelization

Classical .NET parallelization

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

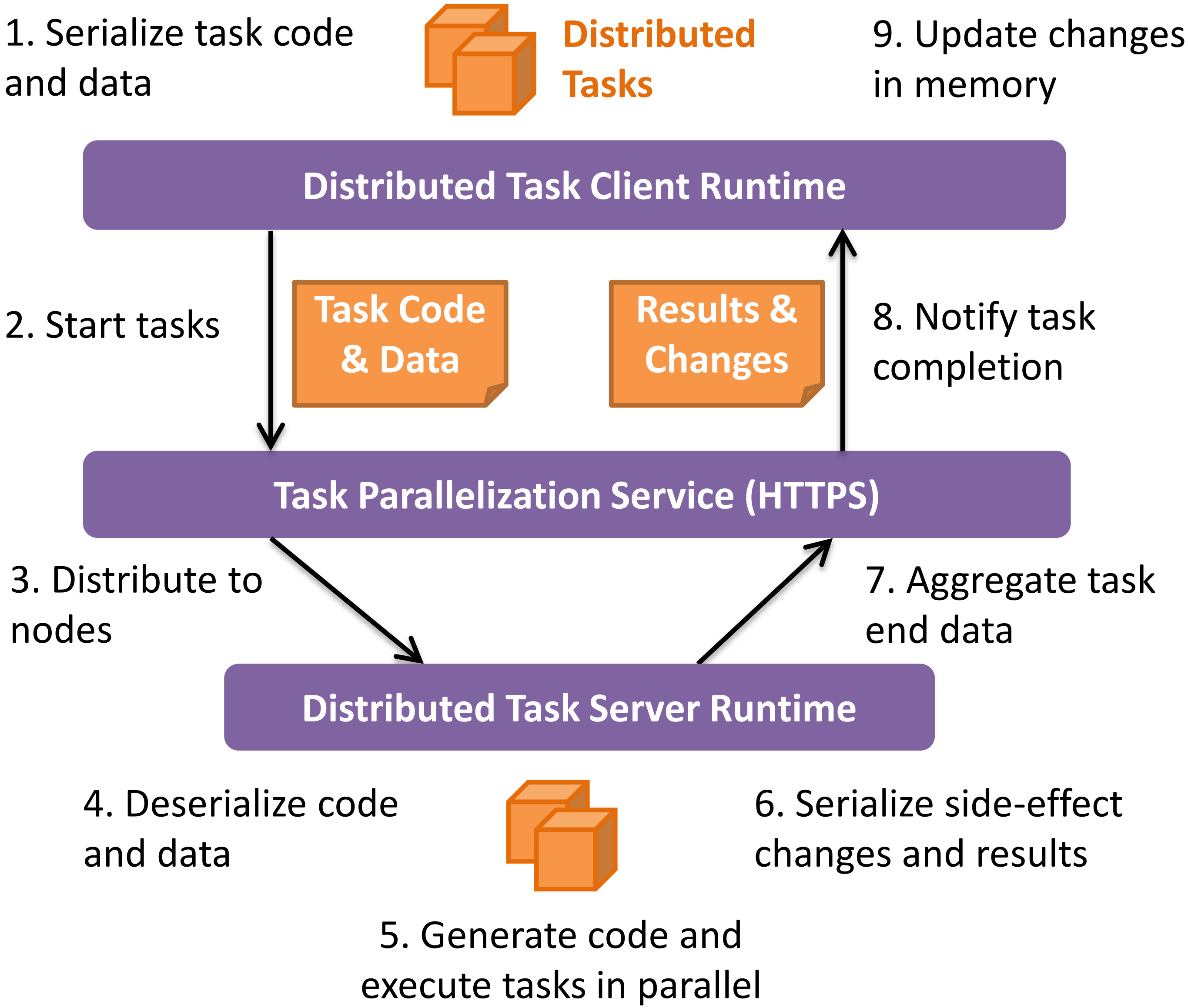
New distributed task parallelization

```
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 can issue side effects (variable changes)
- Tasks must be independent
  - No synchronization => No shared mutable state
  - Embarrassingly parallel => simple and efficient

Runtime System



Task Serialization

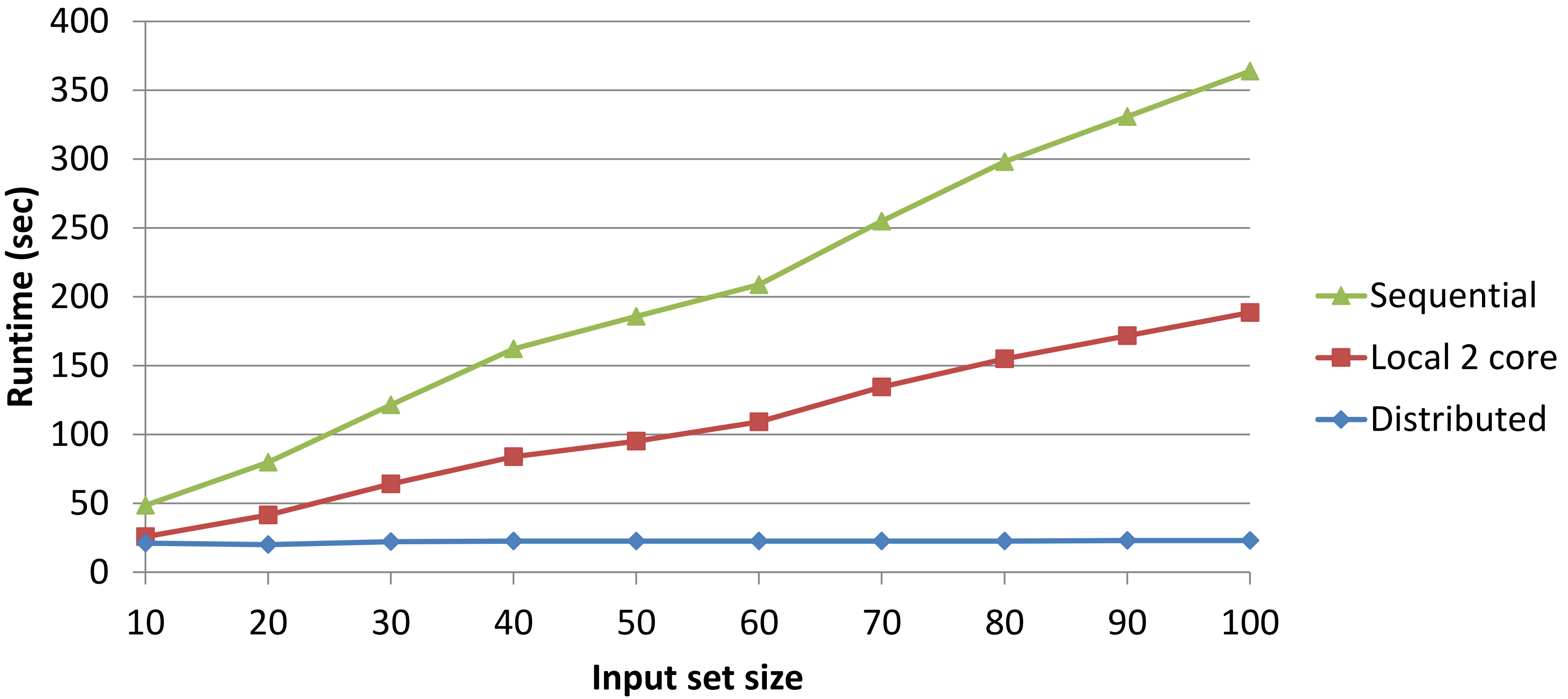
- Potentially executable task code
  - Conservative code analysis
    - Starting from task delegate
    - Directly and indirectly callable methods
    - Potentially used classes and fields
- Potentially accessed task data
  - Partial heap snapshot
    - Graph of reachable objects with accessible fields
    - Accessible static fields / constants
    - Start does not need to block for serialization (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
    - Correct because of task independence
  - Partial data race detection
    - Write/write conflicts between distributed tasks

Performance Scaling

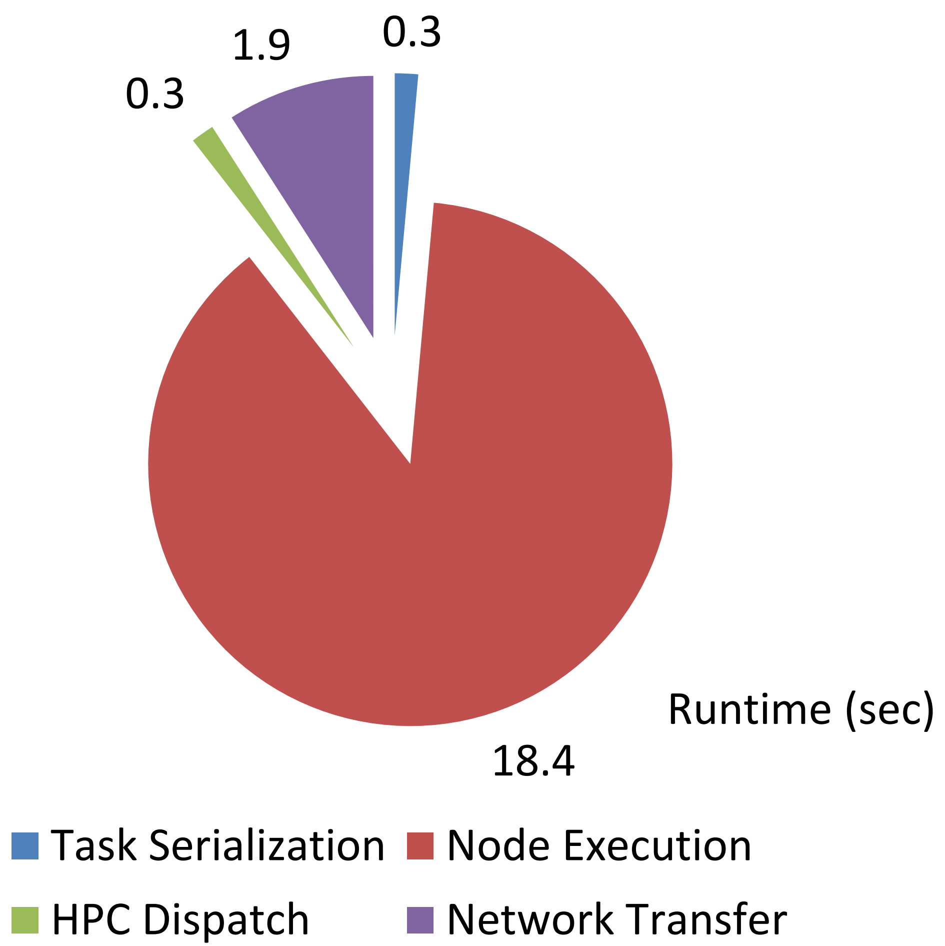
Number factorizations (64 bit, random prime factors around 2^32)



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

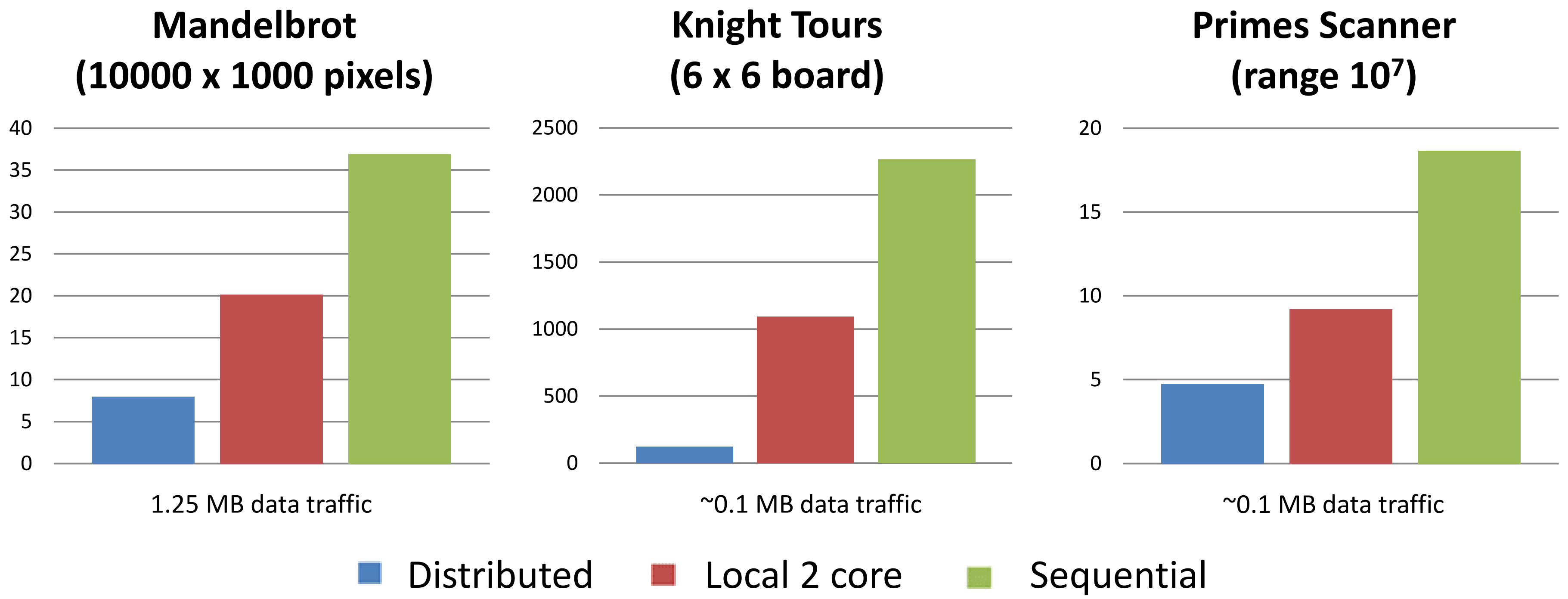
Performance Cost Breakdown

Factorizations (10 numbers)



Performance Comparisons

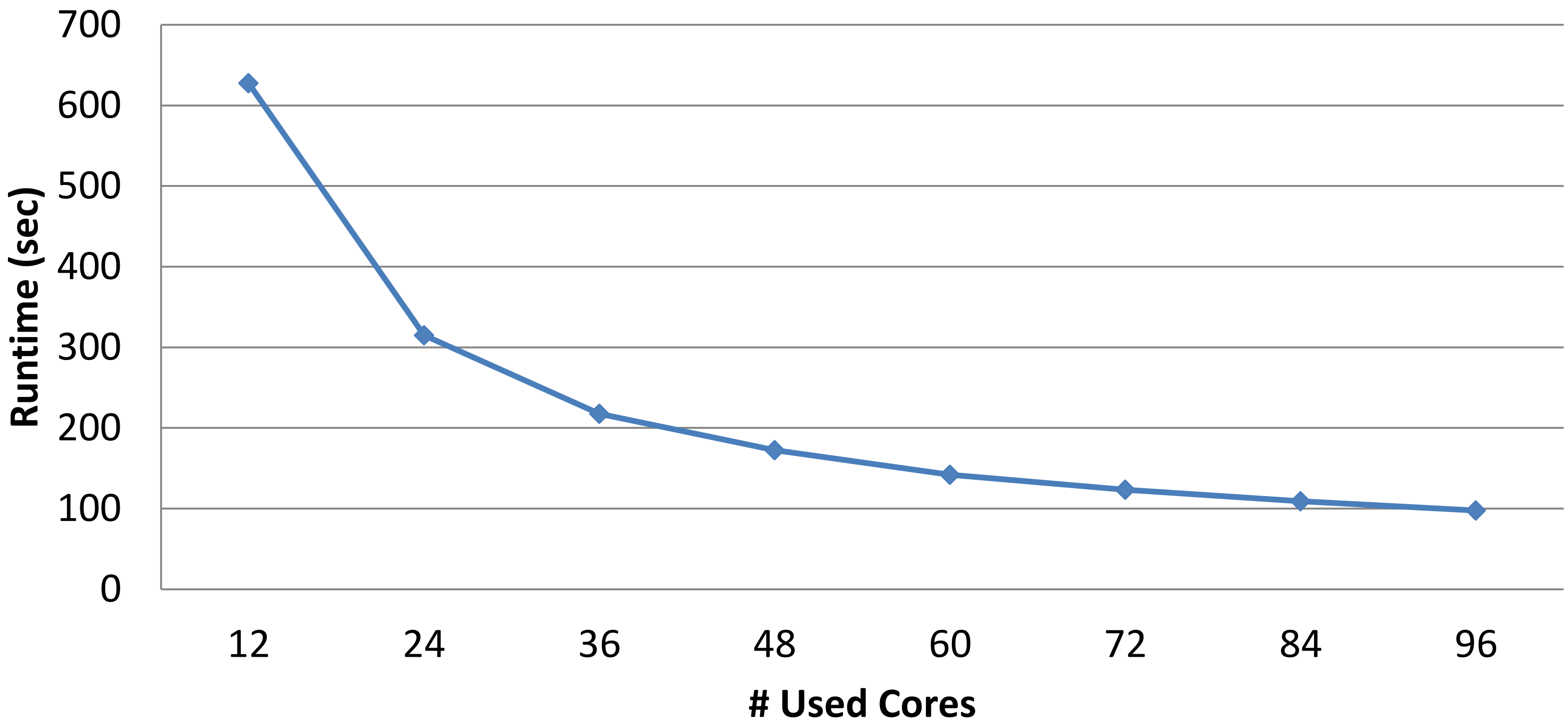
- Three more examples (runtimes in seconds)



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

- Depends on #used cores (factorization)



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

- High parallel speedup possible
- But with inherent overheads
  - Network transmission (throughput + delay)
  - Task serialization / deserialization
  - Dispatching of the HPC cluster job
- Parallelization needs to compensate overheads
  - Compute-intense tasks, relatively small data amount
  - Depending on network / server settings

=> Runtime system itself works efficiently

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 or distribution logic
  - Write/write race detection as extra safeguard
- Future work
  - Task dependencies (chaining)
  - More features, debugging, monitoring

http://concurrency.ch/Projects/TaskParallelism