.NET Task Parallelization as A Service
A Runtime System for Automatic Shared Task Distribution

Luc Bläser
HSR University of Applied Sciences Rapperswil

MULTIPROG2015 @ Hipeac
20 April 2015
Levels of Parallelization

Multi-core

- 2-16 cores
- Inbuilt support in programming language

Multi-processor

- GPU: 512 cores
- Xeon Phi: 61 cores
- Particular distributed programming

Computer Network

- Arbitrary, e.g., 1000 core cluster
- Too seldom used

- Processor
  - Core
  - Core
  - Core
  - Core

- Computer
  - CPU
  - GPU
  - CPU
  - Phi
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

```csharp
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);
}
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

```csharp
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

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

New distributed task parallelization

```csharp
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

1. Serialize task code and data
2. Start tasks
3. Distribute to nodes
4. Deserialize code and data
5. Generate code and execute tasks in parallel
6. Serialize side-effect changes and results
7. Aggregate task end data
8. Notify task completion
9. Update changes in memory

Distributed Task Client Runtime

Task Code & Data

Distributed Tasks

Results & Changes

Task Parallelization Service (HTTPS)

Distributed Task Server Runtime

5. Generate code and execute tasks in parallel
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)

- Task Serialization: 0.3
- Node Execution: 18.4
- HPC Dispatch: 0.3
- Network Transfer: 1.9

Runtime (sec): 18.4
Performance Comparisons

- Three more examples (runtimes in seconds)

**Mandelbrot (10000 x 1000 pixels)**

- Distributed: ~0.1 MB data traffic
- Local 2 core: 1.25 MB data traffic
- Sequential

**Knight Tours (6 x 6 board)**

- Distributed: ~0.1 MB data traffic
- Local 2 core
- Sequential

**Primes Scanner (range 10^7)**

- Distributed: ~0.1 MB data traffic
- Local 2 core
- Sequential

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-intensive 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