Axum

Getting the Genie Back into the Lamp

Niklas Gustafsson
Parallel Computing Platform
Microsoft Corporation
Once upon a time...

My parallel programs looked something like:

```bash
ls -l | awk '{ print "%6d %s\n" NR $0 } | sed 7q
```

People who really knew what they were doing used...

... more complex IPC
... co-routines
... languages with support for pre-emptive multi-tasking
Then...

... C/C++ thread libraries used by experts...
... clock-cycle glass ceiling $\rightarrow$ no more automatic speedups of sequential code...
... multi-core...
... everyone has to leverage parallelism...
... but no one wants to learn a new language...
... a model for experts used by the masses...

Trouble
Shared Memory Parallelism

In theory...
Shared Memory Parallelism

... and practice
Un-partitioned Shared Memory State

Technology scale
Overlapping working sets + false sharing
Cache coherence traffic will dominate

Process scale
Unstated, indirect, subtle data dependencies
Documentation quickly out of sync with reality
Requires enormous discipline
The Web

Loosely coupled

Low trust between "components"

Partitioned state

Message-passing (HTTP Get/Put)
Un-partitioned State

Instead of this...
Partitioned State

... we want this:
Special-purpose language in incubation

Partition data into domains

Agents use shared state within domains

Agents use message-passing between domains

Support for asynchrony and data-flow
Axum Concepts

Domain

Channel

Schema

Agent
Channels

Conduit for messages between two agents
  Support for in-process and distributed messaging

Define ports as communication vocabulary

```plaintext
channel PingPong {
  input bool Ping;
  output Signal Pong;
}
```

Input port ‘Ping,’ sending data from a client to an agent

Output port ‘Pong,’ sending data from an agent to its client
Agents

Active components, similar to threads but has limited access to shared mutable state

Send and receive messages via channels

```csharp
agent PingAgent : channel PingPong
{
public PingAgent ()
{
while (receive(Ping))
{
    Pong <-- Signal.Value;
}
Pong <-- Signal.Value;
}
}
```
Connecting to an Agent

Connections are made in one of two ways:

```csharp
var tbl = provider.Connect<TableAccess>("http://localhost/foo");
```

- A factory for channels
- A channel type
- An agent address

```csharp
var tbl = Table.TableAgent.CreateInNewDomain();
```

- An agent type
- A factory method for in-process agents
Ping Pong

channel PingPong
{
    input bool Ping;
    output Signal Pong;
}

agent PingAgent : channel PingPong
{
    public PingAgent()
    {
        while (receive(Ping))
        {
            Pong <-- Signal.Value;
        }
        Pong <-- Signal.Value;
    }
}

agent MainAgent : channel Microsoft.Axum.Application
{
    public MainAgent()
    {
        var pp = PingAgent.CreateInNewDomain();

        for (int i = 0; i < 100; i++)
        {
            pp::Ping <-- true;
            receive(pp::Pong);
        }

        pp::Ping <-- false;
        receive(pp::Pong);

        Done <-- Signal.Value;
    }
}
Domains

Define state shared by several agent instances
Isolate agents in different domains from each other
Foundation for seamless distribution of agents

Agents are “hosted” within domains

```csharp
domain Table
{
    Dictionary<string, string> dict = new Dictionary<string, string>();

    public Table()
    {
        Host<TableAgent>("TableAgentAddress");
    }

    public agent TableAgent : channel TableAccess ...
}
Domains + Agents = Balanced SM/MP

Agents declared as

- **Writer** – allowed to modify state
- **Reader** – allowed to read mutable state
- **N/A** – no access to mutable state

Agent code is scheduled in parallel based on classification

- Agents in different domains run in parallel
- “One writer, many readers” within each domain
- No-access agents run completely in parallel
Schema

Payload definition

Guaranteed to serialize

Efficient in-process copying

```cpp
schema TableEntry {
    required String Key;
    required String Value;

    rules { require !String.IsNullOrEmpty(Key);
              require !String.IsNullOrEmpty(Value); }
}
```
Protocols

Define legal order of messages

Place constraints on payload values

```
channel TableAccess
{
    input KeyValuePair <String,String> Put;
    input String Get : String;
    input Signal Done;

    Start: { Put $ (!String.IsNullOrEmpty(value.Key)) -> Start;
             Get $ (!String.IsNullOrEmpty(value)) -> Start;
             Done -> End; }
}
```
Dataflow Networks

Propagate data between
  Computations (methods)
  Buffers
  Channel ports

Operators
  forward, forward once, broadcast, alternate, multiplex, join

Operands
  buffer, single-assignment, transforms, sinks

chan::Request ==> TransformString ==> chan::Reply;

buffer -<< { TransformString ==> sink, PrintString };
Ping Pong

agent PingAgent : channel PingPong
{
    public PingAgent()
    {
        Ping ==> (b => Signal.Value) ==> Pong;
    }
}
Summary

• Un-partitioned shared memory state leads to complexity
• Runtime-isolated memory has high overheads
• Static isolation has lower overheads, but still too high
• A mix of static isolation / MP and shared memory balances complexity and performance
• Domain-based isolation provides most of what we need from the programming model to get “transparent” distribution
• Asynchronous interactions become commonplace