Towards Actor Programming for High-Performance Computing

Dominik Charousset
dominik.charousset@haw-hamburg.de

iNET RG, Department of Computer Science
Hamburg University of Applied Sciences

August 2016

Hochschule für Angewandte
Wissenschaften Hamburg
Hamburg University of Applied Sciences

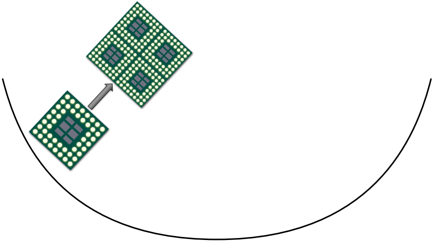
iNET

Concurrency & Beyond

Developers face not one, but multiple trends:

- Concurrency: More cores on desktops & mobiles
- Accelerators: One binary, multiple instruction sets
- Cloud & cluster computing: Highly distributed deployment
- Embedded platforms: Distributed with limited node capabilities

⇒ Heterogeneous platforms, concurrency & distribution

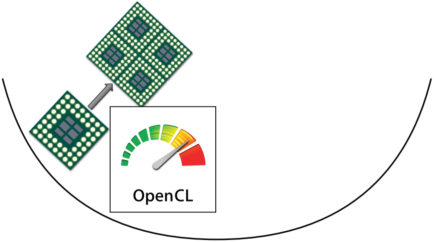


Concurrency & Beyond

Developers face not one, but multiple trends:

- Concurrency: More cores on desktops & mobiles
- Accelerators: One binary, multiple instruction sets
- Cloud & cluster computing: Highly distributed deployment
- Embedded platforms: Distributed with limited node capabilities

⇒ Heterogeneous platforms, concurrency & distribution

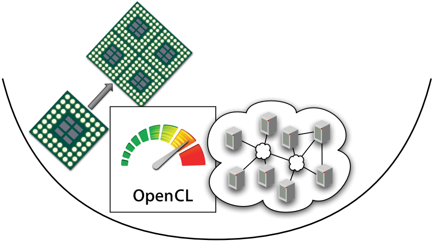


Concurrency & Beyond

Developers face not one, but multiple trends:

- Concurrency: More cores on desktops & mobiles
- Accelerators: One binary, multiple instruction sets
- Cloud & cluster computing: Highly distributed deployment
- Embedded platforms: Distributed with limited node capabilities

⇒ Heterogeneous platforms, concurrency & distribution

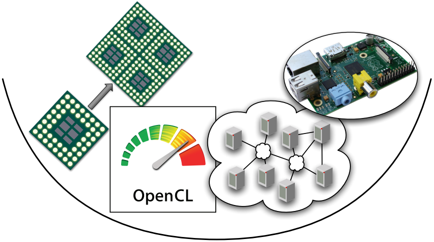


Concurrency & Beyond

Developers face not one, but multiple trends:

- Concurrency: More cores on desktops & mobiles
- Accelerators: One binary, multiple instruction sets
- Cloud & cluster computing: Highly distributed deployment
- Embedded platforms: Distributed with limited node capabilities

⇒ Heterogeneous platforms, concurrency & distribution

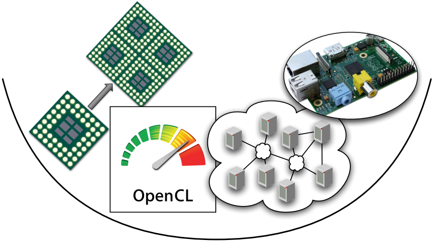


Concurrency & Beyond

Developers face not one, but multiple trends:

- Concurrency: More cores on desktops & mobiles
- Accelerators: One binary, multiple instruction sets
- Cloud & cluster computing: Highly distributed deployment
- Embedded platforms: Distributed with limited node capabilities

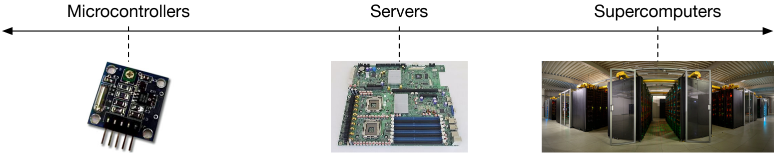
⇒ Heterogeneous platforms, concurrency & distribution



We Need Scalable Abstractions

Programming tools should enable users to scale applications

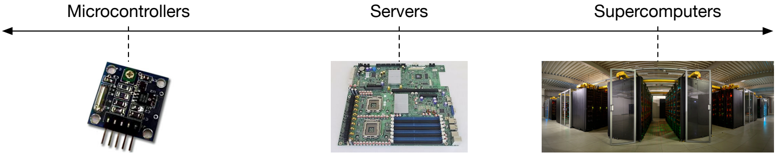
- Avoid race conditions by design (no locks!)
- Keep API stable when transitioning from one to many nodes
- Compose large systems out of small components (testability!)
- Provide a runtime that scales from the IoT up to HPC



We Need Scalable Abstractions

Programming tools should enable users to scale applications

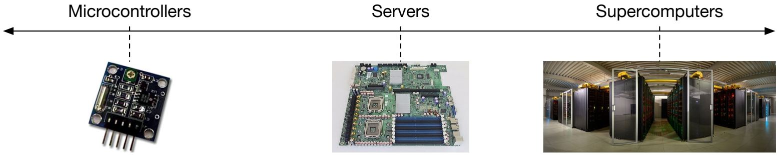
- Avoid race conditions by design (no locks!)
- Keep API stable when transitioning from one to many nodes
- Compose large systems out of small components (testability!)
- Provide a runtime that scales from the IoT up to HPC



We Need Scalable Abstractions

Programming tools should enable users to scale applications

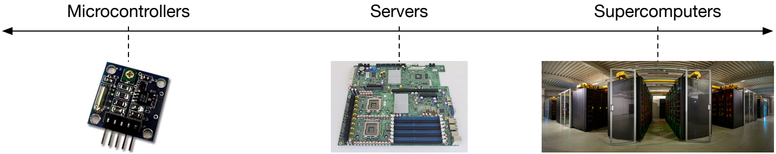
- Avoid race conditions by design (no locks!)
- Keep API stable when transitioning from one to many nodes
- Compose large systems out of small components (testability!)
- Provide a runtime that scales from the IoT up to HPC



We Need Scalable Abstractions

Programming tools should enable users to scale applications

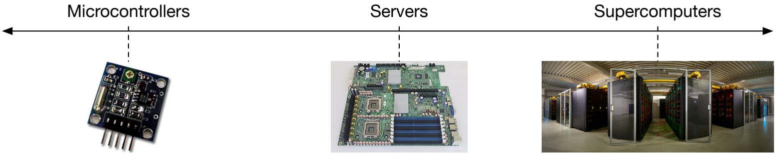
- Avoid race conditions by design (no locks!)
- Keep API stable when transitioning from one to many nodes
- Compose large systems out of small components (testability!)
- Provide a runtime that scales from the IoT up to HPC



We Need Scalable Abstractions

Programming tools should enable users to scale applications

- Avoid race conditions by design (no locks!)
- Keep API stable when transitioning from one to many nodes
- Compose large systems out of small components (testability!)
- Provide a runtime that scales from the IoT up to HPC



Agenda

The Actor Model

Actors are concurrent entities, that ...

- Communicate via message passing
- Do not share state
- Can create ("spawn") more actors
- Can monitor other actors

Brief History of the Actor Model

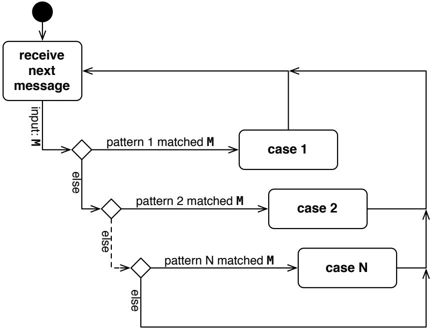
- Term was coined in 1973 by Hewitt, Bishop, and Steiger
- Roots in artificial intelligence & functional programming
- First widespread de facto implementation: Erlang (1986)
- Sudden peak of interest with advent of multicore machines

Implications of the Actor Model

Actor: universal primitive for parallelism

- Actors run concurrently
- Lightweight actors outperform equivalent thread-based approaches
- Actor-based solutions often use divide & conquer strategies
- Deployment at runtime independent from application logic
- Communication between actors is network transparent

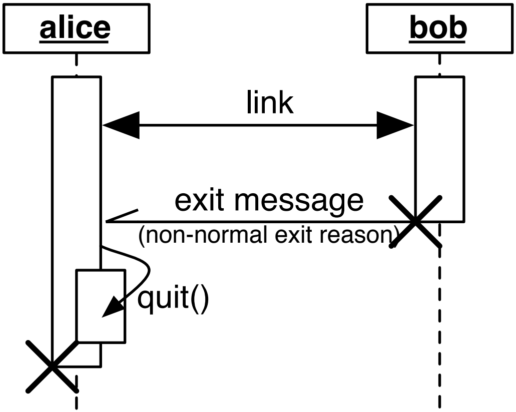
Actor Programming



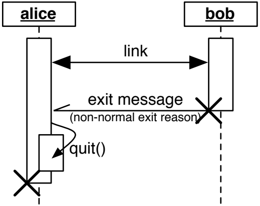
Actor Programming *is* Message-Oriented Programming

- Actors are active objects
- No direct method invocation, only messages
- Messages passing hides location of receiver

Linking Actors



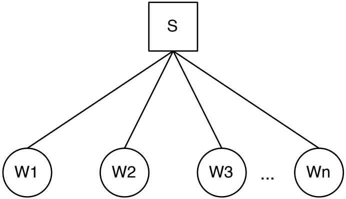
Linking Actors



- Bidirectional monitoring
- Errors are propagated through exit messages
- When receiving an exit message:
  - Actors fail for the same reason per default
  - Actors can override default to handle failures manually
- Build systems where all actors are alive or have collectively failed

Supervision Trees

- High-level hierarchical error management in Erlang
- Upper actors (supervisor) monitor lower actors (workers)
- Policy-based restart of workers (possibly remotely) on failure

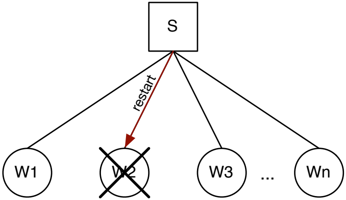


Example tree

1/4

Supervision Trees

- High-level hierarchical error management in Erlang
- Upper actors (supervisor) monitor lower actors (workers)
- Policy-based restart of workers (possibly remotely) on failure

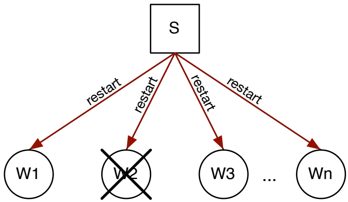


One-for-one strategy

2/4

Supervision Trees

- High-level hierarchical error management in Erlang
- Upper actors (supervisor) monitor lower actors (workers)
- Policy-based restart of workers (possibly remotely) on failure

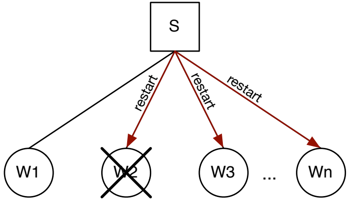


One-for-all strategy

3/4

Supervision Trees

- High-level hierarchical error management in Erlang
- Upper actors (supervisor) monitor lower actors (workers)
- Policy-based restart of workers (possibly remotely) on failure



Rest-for-one strategy

4/4

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction for developing software
- Applies to concurrency *and* distribution
  - Abstraction over deployment
  - Messaging across heterogeneous HW components & networks
- Provides strong failure semantics
  - Hierarchical error management
  - Re-deployment at runtime

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction for developing software
- Applies to concurrency *and* distribution
  - Abstraction over deployment
  - Messaging across heterogeneous HW components & networks
- Provides strong failure semantics
  - Hierarchical error management
  - Re-deployment at runtime

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction for developing software
- Applies to concurrency *and* distribution
  - Abstraction over deployment
  - Messaging across heterogeneous HW components & networks
- Provides strong failure semantics
  - Hierarchical error management
  - Re-deployment at runtime

Actors in High Performance Computing?

- HPC is getting more heterogeneous
  - Accelerators like NVIDIA Tesla & Intel Phi increase complexity
  - MPI and OpenMP only address individual problems at low level
- In-transit and in situ analysis is challenging
  - Low-level messaging APIs come with strong coupling
  - Actors can help to decouple and allow flexible deployment
- Actors systems can move computations instead of data
  - Data storages & networks are bottlenecks
  - Spawning actors near sources increases data locality

Agenda

About CAF

- Lightweight & fast actor model implementation
- First commit: 4 Mar 2011
- Developed at iNET research group (HAW Hamburg)
- $>$ 40,000 lines of code[1]
- Active, international community

[1] https://www.openhub.net/p/actor-framework

Who is using CAF?

CAF currently focuses on infrastructure software

- Building blocks for essential software components
- Emphasis on reliability, efficiency and maintainability
- Relevant to users in both academia and industry

CAF in MMOs

Dual Universe[2]

- Single Shard Sandbox MMO
- Backend based on CAF
- Developed at Novaquark (Paris), currently in pre-alpha

[2] http://www.dual-thegame.com/

CAF in Network Forensics

VAST: Visibility Across Space and Time[3]

- Platform for network forensics and incident response
- Distributed realtime indexing of network events
- Interactive, iterative queries on large data sets
- Developed at UC Berkeley, currently in pre-alpha

[3] http://vast.io

CAF in Communication Backends

Broker[4]: Bro's Messaging Library

- Implements Bro's high-level communication patterns
- Subscription-based communication model
- Distributed data store and event handling
- Developed at ICSI Berkeley, currently in beta

[4] https://github.com/bro/broker

CAF Actors are meant to Scale

- Flexible C++ runtime that adapts to deployment
  - Scale up/down from motes to high-end servers
  - Scale in/out from desktops to clusters
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Transparent integration of OpenCL-based actors
  - Hide complexity of communicating with heterogeneous hardware
  - Offload work to GPGPUs simply by sending messages

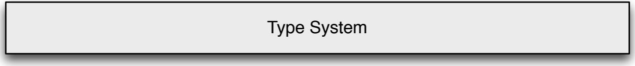
CAF Actors are meant to Scale

- Flexible C++ runtime that adapts to deployment
  - Scale up/down from motes to high-end servers
  - Scale in/out from desktops to clusters
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Transparent integration of OpenCL-based actors
  - Hide complexity of communicating with heterogeneous hardware
  - Offload work to GPGPUs simply by sending messages

CAF Actors are meant to Scale

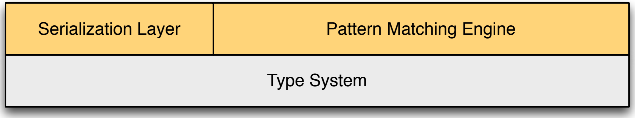
- Flexible C++ runtime that adapts to deployment
  - Scale up/down from motes to high-end servers
  - Scale in/out from desktops to clusters
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Transparent integration of OpenCL-based actors
  - Hide complexity of communicating with heterogeneous hardware
  - Offload work to GPGPUs simply by sending messages

Core Architecture of CAF



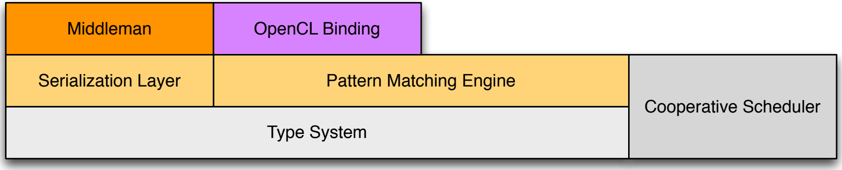
1/8

Core Architecture of CAF



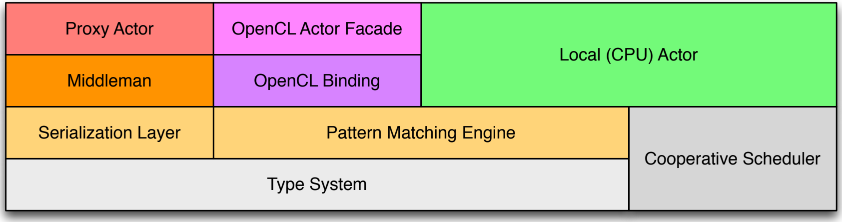
2/8

Core Architecture of CAF

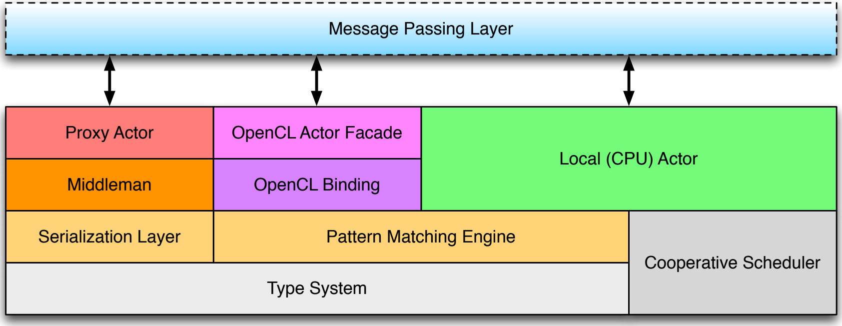


3/8

Core Architecture of CAF

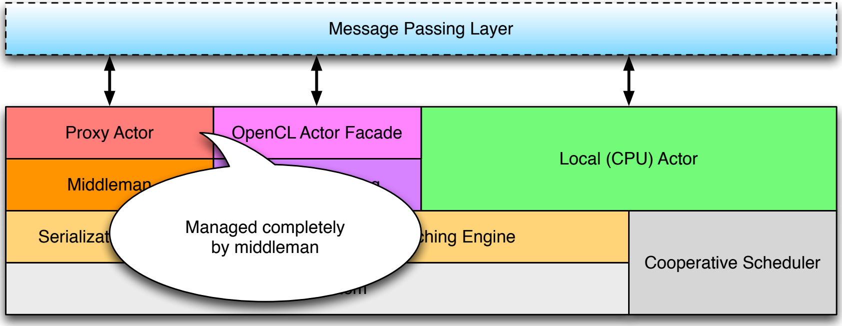


Core Architecture of CAF



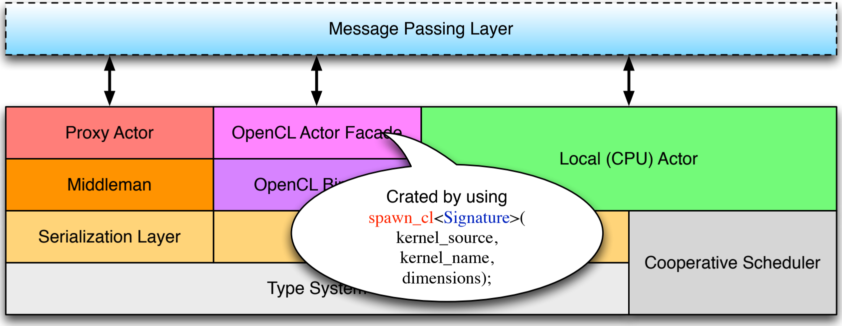
5/8

Core Architecture of CAF



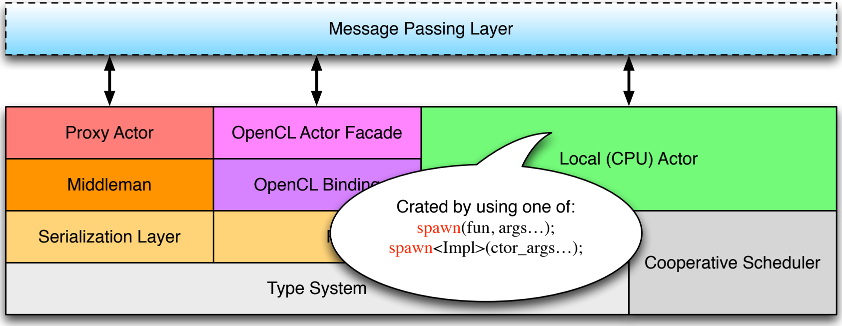
6/8

Core Architecture of CAF



7/8

Core Architecture of CAF



8/8

Class vs. Actor Interfaces

```
class KeyValStore {
public:
  void set(Key k, Val v);
  Val get(Key k) const;
};
```

```
using KeyValStore =
 typed_actor<
  reacts_to<set, Key, Val>,
  replies_to<get, Key>
      ::with<Val>>;
```

- Method invocation
- Race conditions likely
- Concurrent performance is a function of developer skill

- Message passing
- Data race impossible
- Supports massively parallel access & remote invocation

Class vs. Actor Interfaces

```
class KeyValStore {
public:
  void set(Key k, Val v);
  Val get(Key k) const;
};
```

- Method invocation
- Race conditions likely
- Concurrent performance is a function of developer skill

```
using KeyValStore =
 typed_actor<
  reacts_to<set, Key, Val>,
  replies_to<get, Key>
      ::with<Val>>;
```

- Message passing
- Data race impossible
- Supports massively parallel access & remote invocation

Class vs. Actor Interfaces

```
class KeyValStore {
public:
  void set(Key k, Val v);
  Val get(Key k) const;
};
```

- Method invocation
- Race conditions likely
- Concurrent performance is a function of developer skill

```
using KeyValStore =
 typed_actor<
  reacts_to<set, Key, Val>,
  replies_to<get, Key>
      ::with<Val>>;
```

- Message passing
- Data race impossible
- Supports massively parallel access & remote invocation

Class vs. Actor Interfaces

```
class KeyValStore {
public:
  void set(Key k, Val v);
  Val get(Key k) const;
};
```

- Method invocation
- Race conditions likely
- Concurrent performance is a function of developer skill

```
using KeyValStore =
 typed_actor<
  reacts_to<set, Key, Val>,
  replies_to<get, Key>
       ::with<Val>>;
```

- Message passing
- Data race impossible
- Supports massively parallel access & remote invocation

Simple Calculator in CAF

```
using math_t = typed_actor<replies_to<int,int>::with<int>>;
math_t::behavior_type math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  };
}
void math_client(event_based_actor* self, math_t ms) {
  self->request(ms, 10s, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// system.spawn(math_client, system.spawn(math_server));
```

Simple Calculator in CAF

```
using math_t = typed_actor<replies_to<int,int>::with<int>>;
math_t::behavior_type math_server() {
  return {
    [](int a, int b) {
      return a +
    }
  };
}
void math_client(event_based_actor* self, math_t ms) {
  self->request(ms, 10s, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// system.spawn(math_client, system.spawn(math_server));
```

typedef with interface definition for convenience

Simple Calculator in CAF

```
using math_t = typed_actor<replies_to<int,int>::with<int>>;
math_t::behavior_type math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  };
}
voi                                           self, math_t ms) {
  se                                         n(
    
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// system.spawn(math_client, system.spawn(math_server));
```

return message handler for incoming messages (used until replaced or actor is done)

Simple Calculator in CAF

```
using math_t = typed_actor<replies_to<int,int>::with<int>>;
math_t::behavior_type math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  ...
}
vo...                                  * self, math_t ms) {
  self->request(ms, 10s, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// system.spawn(math_client, system.spawn(math_server));
```

types of message handlers
must match interface definition

Simple Calculator in CAF

```
using math_t = typed_actor<replies_to<int,int>::with<int>>;
math_t::behavior_type math_server() {
  return {
    [](int
      retur
    }
  };
}
void math_client(event_based_actor* self, math_t ms) {
  self->request(ms, 10s, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// system.spawn(math_client, system.spawn(math_server));
```

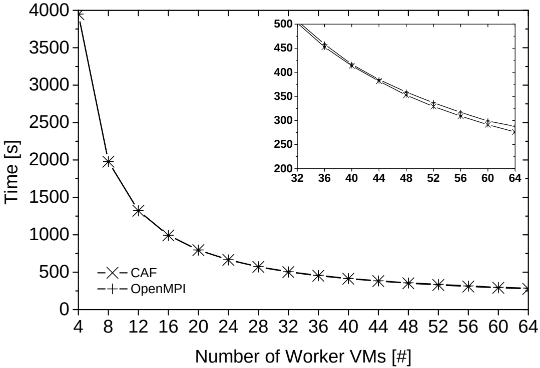
send a message and then wait for response (using a "one-shot handler")

Agenda

Case Study: Distributed Mandelbrot

- Calculate images of the Mandelbrot set in C++
- Simple, highly distributable algorithm
- Distributed using (1) CAF and (2) OpenMPI
  - Same source code for calculation
  - Only the message passing layers differ

Case Study Results



- Both implementations exhibit equal scaling behavior
- Doubling the number of worker nodes halves the runtime
- CAF slightly faster despite higher level of abstraction

Case Study Discussion

- Higher level of abstraction does not imply lower performance
- Message passing in CAF has minimal overhead
- Relatively small number of nodes in test setup (64)
- No other MPI implementations tested (e.g. Intel MPI)
- CAF not yet ready for HPC technologies such as Infiniband

Agenda

Goal: Data-driven Actors

- User defines high-level data model
- Runtime instantiates actors on demand
- Self-organizing runtime instead of hard-wired messaging
- Maximize data locality by moving actors, not data
- Build extensible message flows and computation steps
- Tools for visualization of system state & live data

Benefits for HPC Developers

- Reduced complexity
- Faster development cycles
- Domain-specific data models
- Composability and re-usability

Challenges for CAF

- HPC network backend (Infiniband)
- DSL for defining data models and task dependencies
- New tools for visualization and debugging
- Optimize CAF internals further, particularly memory access
- Scalable algorithms for self-organizing actors

Agenda

Improve Existing HPC Tool

- Parallel actor pipelines as first step
- Model data processing steps as actors
- Gain experience with "less critical" post processing
- Example: parallel Climate Data Operators (CDO) with CAF

Integrate CAF into Existing Middleware

- Deploy actors in data streams of Net-CDF4/HDF5
- Re-use data processing actors for in-transit operations
- Proof of concept for dynamic in-transit work flows with CAF

Agenda

Conclusion

- We believe actor programming is a good fit for HPC
- A high level of abstraction can increase robustness & efficiency
- Data-driven actors are also relevant in smaller deployments
- The most important first step for us is finding the right community

Conclusion

- We believe actor programming is a good fit for HPC
- A high level of abstraction can increase robustness & efficiency
- Data-driven actors are also relevant in smaller deployments
- The most important first step for us is finding the right community

Conclusion

- We believe actor programming is a good fit for HPC
- A high level of abstraction can increase robustness & efficiency
- Data-driven actors are also relevant in smaller deployments
- The most important first step for us is finding the right community

Conclusion

- We believe actor programming is a good fit for HPC
- A high level of abstraction can increase robustness & efficiency
- Data-driven actors are also relevant in smaller deployments
- The most important first step for us is finding the right community

Thank you for your attention!

Home page: http://actor-framework.org

iNET research group: http://inet.cpt.haw-hamburg.de

Sources: https://github.com/actor-framework/actor-framework