Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments

Dominik Charousset
dominik.charousset@haw-hamburg.de

iNET RG, Department Informatik
HAW Hamburg

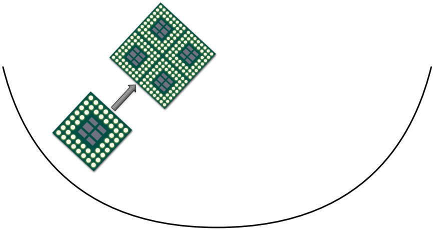
Oct 2013

Agenda

Challenges of Modern Systems

Developers face not one, but multiple trends:

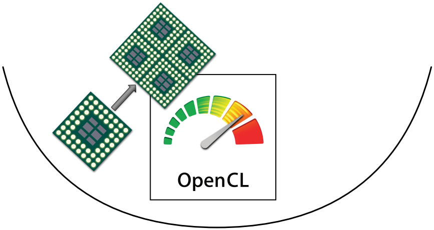
- More cores on both desktop & mobile plattforms



Challenges of Modern Systems

Developers face not one, but multiple trends:

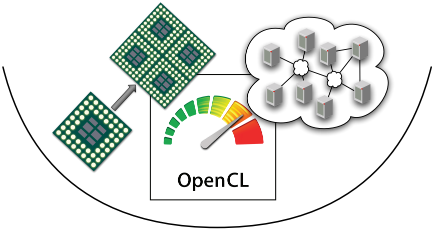
- More cores on both desktop & mobile plattforms
- SIMD components: GPUs can vastly outperform CPUs



Challenges of Modern Systems

Developers face not one, but multiple trends:

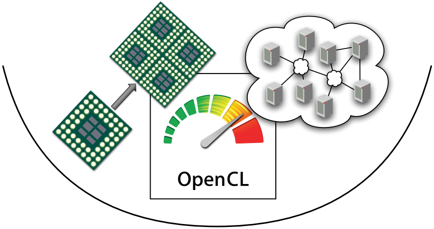
- More cores on both desktop & mobile plattforms
- SIMD components: GPUs can vastly outperform CPUs
- Cloud computing: "Infrastructure as a service"



Challenges of Modern Systems

Developers face not one, but multiple trends:

- More cores on both desktop & mobile plattforms
- SIMD components: GPUs can vastly outperform CPUs
- Cloud computing: "Infrastructure as a service"
- Heterogeneous Environments: From motes to high-end servers

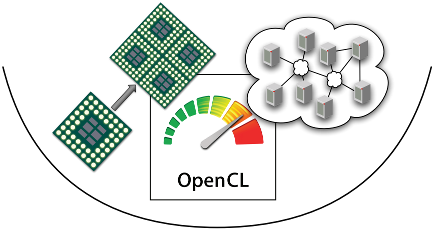


Challenges of Modern Systems

Developers face not one, but multiple trends:

- More cores on both desktop & mobile plattforms
- SIMD components: GPUs can vastly outperform CPUs
- Cloud computing: "Infrastructure as a service"
- Heterogeneous Environments: From motes to high-end servers

⇒ Parallelization, specialization & distribution



Performance & Composability

In order to make use of parallel hardware, we need to ...

- Split application logic into many tasks
- Minimize overhead for launching tasks and collecting results

Performance & Composability

In order to make use of parallel hardware, we need to ...

- Split application logic into many tasks
- Minimize overhead for launching tasks and collecting results

In order to compose systems, we need to ...

- Make use of distributed & heterogeneous resources
- Collect results transparently

Performance & Composability

In order to make use of parallel hardware, we need to ...

- Split application logic into many tasks
- Minimize overhead for launching tasks and collecting results

In order to compose systems, we need to ...

- Make use of distributed & heterogeneous resources
- Collect results transparently

$\Rightarrow$ Late binding of software components to resources

Agenda

The Problem With Implicit Sharing

- Implicit sharing is still the dominant programming model
- Multiple threads can share objects in process-wide memory
- Concurrent access to stateful objects needs synchronization
- Challenges are ...

The Problem With Implicit Sharing

- Implicit sharing is still the dominant programming model
- Multiple threads can share objects in process-wide memory
- Concurrent access to stateful objects needs synchronization
- Challenges are ...
  - Race conditions ("solved" by locks)
  - Deadlocks/Lifelocks (caused by locks)
  - Poor scalability due to queueing (Coarse-Grained Locking)
  - High complexity (Fine-Grained Locking)

The Problem With Implicit Sharing

- Implicit sharing is still the dominant programming model
- Multiple threads can share objects in process-wide memory
- Concurrent access to stateful objects needs synchronization
- Challenges are ...
  - Race conditions (“solved” by locks)
  - Deadlocks/Lifelocks (caused by locks)
  - Poor scalability due to queueing (Coarse-Grained Locking)
  - High complexity (Fine-Grained Locking)
- Locks are not composable

Agenda

The Actor Model

Actors are concurrent entities, that …

- Communicate via message passing
- Do not share state
- Can create ("spawn") new actors
- Can monitor other actors
- Can be freely distributed

Benefits of the Actor Model

- High-level, explicit communication: no locks, no implicit sharing
- Applies to both concurrency *and* distribution
  - Divide workload by spawning actors
  - Network-transparent messaging
- Known to provide strong failure semantics (e.g. Erlang)
- A lightweight implementation allows millions of active actors

Current Limitations of the Actor Model

- Actors have not yet entered the native programming domain

Current Limitations of the Actor Model

- Actors have not yet entered the native programming domain
- Original actor model not ready for Internet scale
  - Loosely coupled orchestration missing
  - No semantics for contacting unknowns
  - 1:1 communication only, no publish/subscribe layer
  - Security model for loosely coupled systems undefined

Current Limitations of the Actor Model

- Actors have not yet entered the native programming domain
- Original actor model not ready for Internet scale
  - Loosely coupled orchestration missing
  - No semantics for contacting unknowns
  - 1:1 communication only, no publish/subscribe layer
  - Security model for loosely coupled systems undefined
- Actor systems need to include heterogeneous components
  - Lack of GPGPU programming support
  - No transparent integration of specialized HW components

Current Limitations of the Actor Model

- Actors have not yet entered the native programming domain
- Original actor model not ready for Internet scale
  - Loosely coupled orchestration missing
  - No semantics for contacting unknowns
  - 1:1 communication only, no publish/subscribe layer
  - Security model for loosely coupled systems undefined
- Actor systems need to include heterogeneous components
  - Lack of GPGPU programming support
  - No transparent integration of specialized HW components
- Actor systems not available for embedded systems

`libcppa` – Actors in C++11

- `libcppa` is an actor system for C++11

`libcppa` – Actors in C++11

- `libcppa` is an actor system for C++11
- Internal DSL for pattern matching of messages

libcppa – Actors in C++11

- libcppa is an actor system for C++11
- Internal DSL for pattern matching of messages
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation

`libcppa` – Actors in C++11

- `libcppa` is an actor system for C++11
- Internal DSL for pattern matching of messages
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Targets both low-end and high-performance computing
  - Embedded HW, e.g., running RIOT
  - Server systems & cluster

libcppa – Actors in C++11

- libcppa is an actor system for C++11
- Internal DSL for pattern matching of messages
- Efficient program execution
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Targets both low-end and high-performance computing
  - Embedded HW, e.g., running RIOT
  - Server systems & cluster
- Transparent integration of OpenCL-based actors

Classes vs. Actors

```cpp
class KeyValStore {
public:

  void set(Key k, Val v);
  Val get(Key k) const;
};
```

Classes vs. Actors

```
class KeyValStore {
public:

  void set(Key k, Val v);
  Val get(Key k) const;
};
```

```
become (
 on(atom("set"), arg_match)
 >> [=](Key k, Val v) { },
 on(atom("get"), arg_match)
 >> [=](Key k) { }
);
```

Classes vs. Actors

```cpp
class KeyValStore {
public:

  void set(Key k, Val v);
  Val get(Key k) const;
};
```

- Method invocation

```cpp
become (
 on(atom("set"), arg_match)
 >> [=](Key k, Val v) { },
 on(atom("get"), arg_match)
 >> [=](Key k) { }
);
```

- Message passing

Classes vs. Actors

```cpp
class KeyValStore {
public:

  void set(Key k, Val v);
  Val get(Key k) const;
};
```

- Method invocation
- Race conditions likely

```cpp
become (
 on(atom("set"), arg_match)
 >> [=](Key k, Val v) { },
 on(atom("get"), arg_match)
 >> [=](Key k) { }
);
```

- Message passing
- Data race impossible

Classes vs. Actors

```cpp
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

```cpp
become (
 on(atom("set"), arg_match)
 >> [=](Key k, Val v) { },
 on(atom("get"), arg_match)
 >> [=](Key k) { }
);
```

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

Agenda

Measurements

Benchmarks are based on the following implementations:

- `cppa` C++ (GCC 4.7.2) with libcppa
- `scala` Scala 2.10 with the Akka library
- `erlang` Erlang 5.9.1

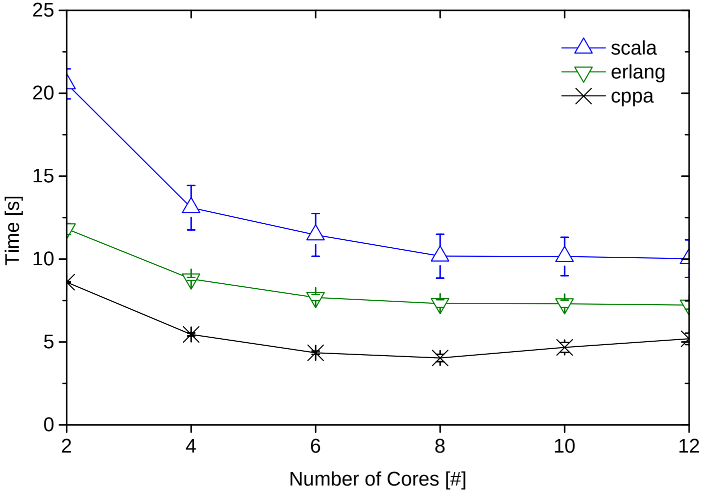
System setup:

- Two hexa-core Intel Xeon 2.27 GHz
- JVM configured with a maximum of 4 GB of RAM
- We vary the number of CPU cores from 2 to 12

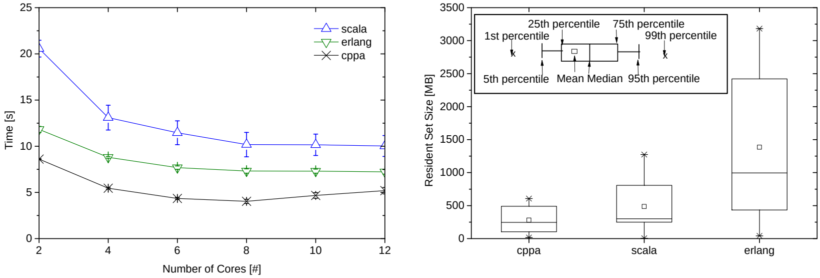
Overhead of Actor Creation

- Fork/join workflow to compute 2^N
  - Each fork step spawns two new actors
  - Join step sums up messages from children
  - Each actor at the leaf sends 1 to parent
- Benchmark creates $\approx$ 1,000,000 actors ($N = 20$)

Overhead of Actor Creation



Overhead of Actor Creation

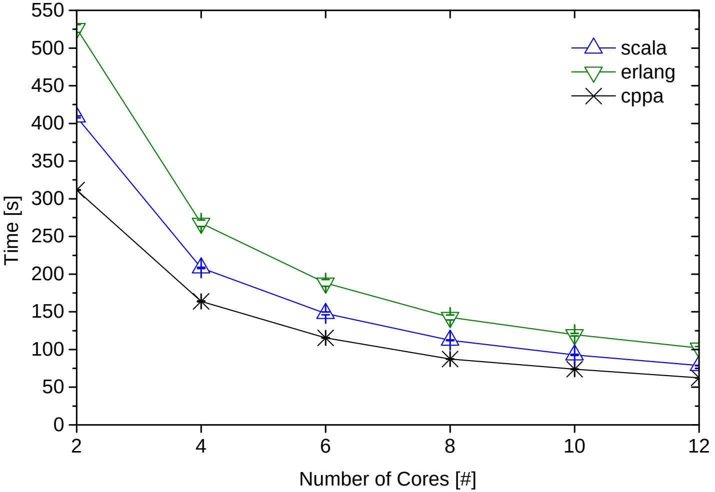


- All three implementations scale up to large actor systems
- Scala and Erlang remain almost constant from 8 cores onwards
- `libcppa` performs best, but slows down after 8 cores

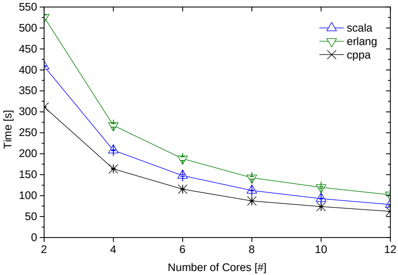
Performance in a Mixed Scenario

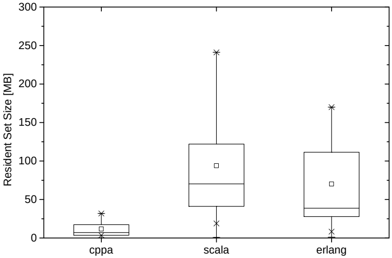
- Mixed operations under work load
- 20 rings of 50 actors each
- Token-forwarding on each ring until 1,000 iterations are reached
- 20 re-creations per ring
- One prime factorization per (re)-created ring to add work load

Performance in a Mixed Scenario



Performance in a Mixed Scenario





- All three implementations exhibit comparable scaling behavior
- JVM performs compute-intensive tasks faster than Erlang's VM
  - Tail-recursive prime factorization in Scala as fast as C++ version
- `libcppa` performs best & uses significantly fewer memory

Matrix Multiplication

- Simple multiplication algorithm using three nested loops
- Implemented
  - Using threads
  - Using actors
  - Using an OpenCL kernel
- C++ implementation is parallelized on the most inner loop
  - Creates $Rows \cdot Columns$ threads or actors

Matrix Multiplication

Setup: 12 cores, Linux, 1000x1000 matrices

Single-threaded	9.029 s
Actors	
OpenCL	
Threads	

Matrix Multiplication

Setup: 12 cores, Linux, 1000x1000 matrices

Single-threaded	9.029 s
Actors	2.428 s
OpenCL	
Threads	

Matrix Multiplication

Setup: 12 cores, Linux, 1000x1000 matrices

Single-threaded	9.029 s
Actors	2.428 s
OpenCL	0.288 s
Threads	

Matrix Multiplication

Setup: 12 cores, Linux, 1000x1000 matrices

Single-threaded	9.029 s	
Actors	2.428 s	
OpenCL	0.288 s	
Threads	…	exception: "`std::system_error`"; per default, 1M threads are not supported

Matrix Multiplication

Setup: 12 cores, Linux, 1000x1000 matrices

Single-threaded	9.029 s	
Actors	2.428 s	
OpenCL	0.288 s	
Threads	...	exception: “`std::system_error`”; per default, 1M threads are not supported

- Threads do not scale up to large numbers
- Number of actors only limited by available memory

Agenda

Conclusion

State of `libcppa`:

- Open source (GPLv2) in Version 0.7
- Hosted on GitHub since Mar 04, 2011
- Runs on GCC $\geq$ 4.7 and Clang $\geq$ 3.2 (Linux & Mac)
- Offers initial support for publish/subscribe communication
- Integrates OpenCL by creating actors from OpenCL kernels

Conclusion

State of `libcppa`:

- Open source (GPLv2) in Version 0.7
- Hosted on GitHub since Mar 04, 2011
- Runs on GCC $\geq$ 4.7 and Clang $\geq$ 3.2 (Linux & Mac)
- Offers initial support for publish/subscribe communication
- Integrates OpenCL by creating actors from OpenCL kernels

Deployment:

- Cooperation with UC Berkeley (research group of Vern Paxson)
  - Actor-based realtime intrusion detection system
- Ongoing negotiation to bundle `libcppa` with Boost libraries
- Currently porting `libcppa` to ARM & embedded systems

Open Research Questions

- Distributed scheduling & load balancing
  - Can one derive migration strategies from communication patterns?
  - How to design a distributed workload management for actors?

Open Research Questions

- Distributed scheduling & load balancing
  - Can one derive migration strategies from communication patterns?
  - How to design a distributed workload management for actors?
- Loosely coupled communication scenarios for actors
  - How to define a scalable publish/subscribe layer for actors?
  - How to orchestrate multiple independent actor systems?
  - Which security design is appropriate for loosely coupled actors?
  - How to propagate errors in non-hierarchical actor systems?

Open Research Questions

- Distributed scheduling & load balancing
  - Can one derive migration strategies from communication patterns?
  - How to design a distributed workload management for actors?
- Loosely coupled communication scenarios for actors
  - How to define a scalable publish/subscribe layer for actors?
  - How to orchestrate multiple independent actor systems?
  - Which security design is appropriate for loosely coupled actors?
  - How to propagate errors in non-hierarchical actor systems?
- Message routing & composability
  - How to define efficient routing of messages?
  - How to process or transform types in in routed messages?
  - How should errors be handled & reported?

Publications

Dominik Charousset, Sebastian Meiling, Thomas C. Schmidt, and Matthias Wählisch.
A Middleware for Transparent Group Communication of Globally Distributed Actors.
In *Middleware Posters 2011*, New York, USA, Dec. 2011. ACM, DL.

Dominik Charousset, Thomas C. Schmidt, and Matthias Wählisch.
Actors and Publish/Subscribe: An Efficient Approach to Scalable Distribution in Data Centers.
In *Proc. of the ACM SIGCOMM CoNEXT. Student Workshop*, New York, Dec. 2012. ACM.

Dominik Charousset and Thomas C. Schmidt.
libcppa - Designing an Actor Semantic for C++11.
In *Proc. of C++Now*, 2013.

Thank you for your attention!

Developer blog: http://libcppa.org

Sources: https://github.com/Neverlord/libcppa

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

Multiply Matrices

```
static constexpr size_t matrix_size = /*...*/;

// always rows == columns == matrix_size
class matrix {
 public:
  float& operator()(size_t row, size_t column);
  const vector<float>& data() const;
  // ...
 private:
  vector<float> m_data; // glorified vector
};
```

Multiply Matrices – Simple Loop

```
matrix simple_multiply(const matrix& lhs,
                       const matrix& rhs) {
  matrix result;
  for (size_t r = 0; r < matrix_size; ++r) {
    for (size_t c = 0; c < matrix_size; ++c) {
      // each calculation can run independently
      result(r, c) = dot_product(lhs, rhs, r, c);
    }
  }
  return move(result);
}
```

Multiply Matrices – std::async

```
matrix async_multiply(const matrix& lhs,
                      const matrix& rhs) {
  matrix result;
  vector<future<void>> futures;
  futures.reserve(matrix_size * matrix_size);
  for (size_t r = 0; r < matrix_size; ++r) {
    for (size_t c = 0; c < matrix_size; ++c) {
      futures.push_back(async(launch::async, [&,r,c] {
        result(r, c) = dot_product(lhs, rhs, r, c);
      }));
    }
  }
  for (auto& f : futures) f.wait();
  return move(result);
}
```

Multiply Matrices – libcppa Actors

```
matrix actor_multiply(const matrix& lhs,
                      const matrix& rhs) {
  matrix result;
  for (size_t r = 0; r < matrix_size; ++r) {
    for (size_t c = 0; c < matrix_size; ++c) {
      spawn([&,r,c] {
        result(r, c) = dot_product(lhs, rhs, r, c);
      });
    }
  }
  await_all_others_done();
  return move(result);
}
```

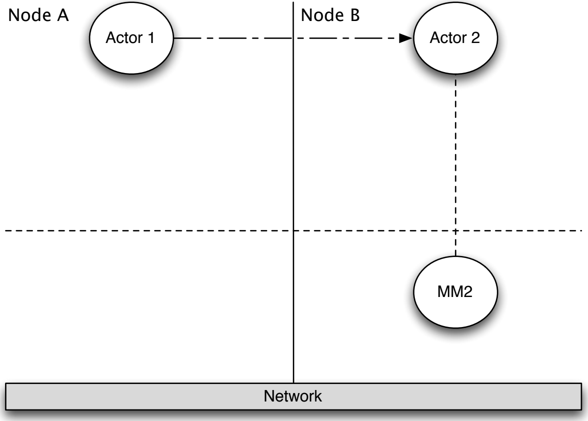
Multiply Matrices – OpenCL Actors

```
static constexpr const char* source = R"__(
  __kernel void multiply(__global float* lhs,
                         __global float* rhs,
                         __global float* result) {
    size_t size = get_global_size(0);
    size_t r = get_global_id(0);
    size_t c = get_global_id(1);
    float dot_product = 0;
    for (size_t k = 0; k < size; ++k)
      dot_product += lhs[k+c*size] * rhs[r+k*size];
    result[r+c*size] = dot_product;
  }
)__";
```

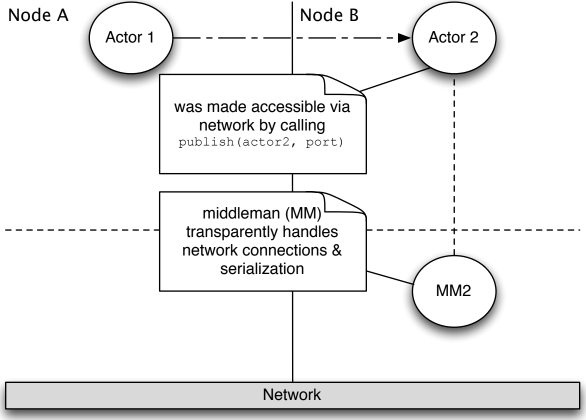
Multiply Matrices – OpenCL Actors

```
matrix opencl_multiply(const matrix& lhs,
                       const matrix& rhs) {
                         // function signature
  auto worker = spawn_cl<float* (float* ,float*)>(
                  // code, kernel name & dimensions
                  source, "multiply",
                  {matrix_size, matrix_size});
  // ordinary message passing
  send(worker, lhs.data(), rhs.data());
  matrix result;
  receive(on_arg_match >> [&](fvec& res_vec) {
    result = move(res_vec);
  });
  return move(result);
}
```

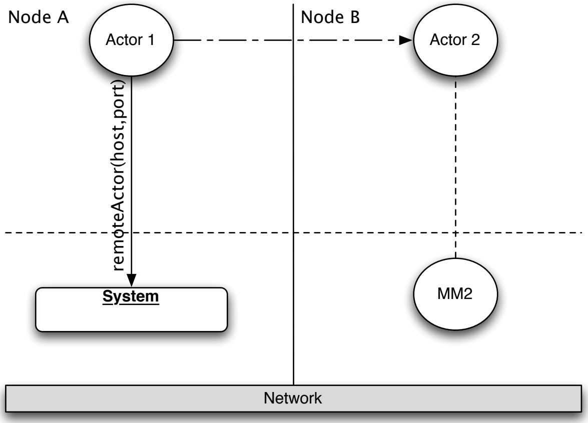
Network Transparency



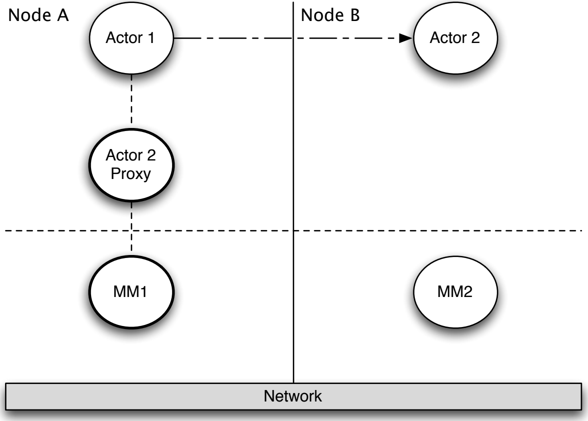
Network Transparency



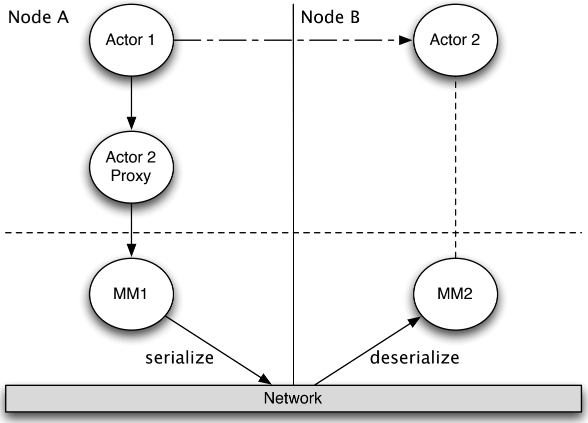
Network Transparency



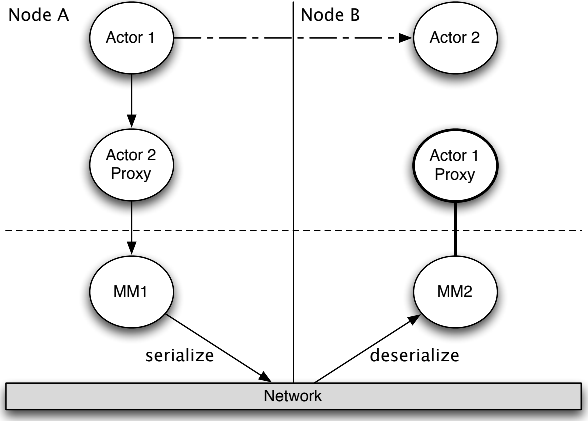
Network Transparency



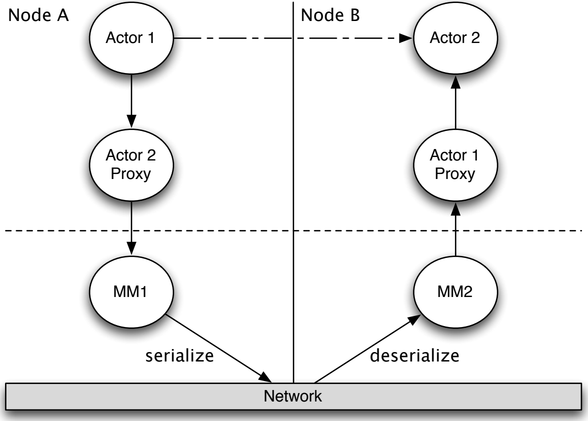
Network Transparency



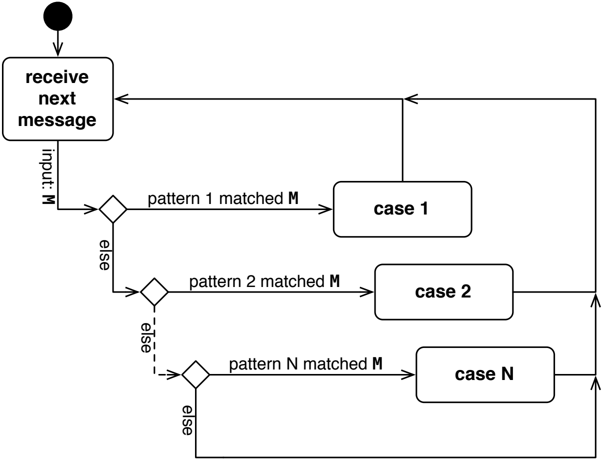
Network Transparency



Network Transparency

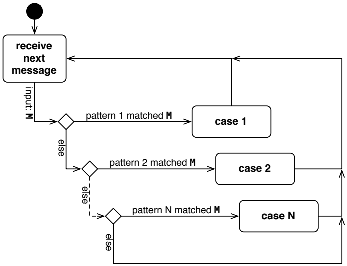


Message Processing



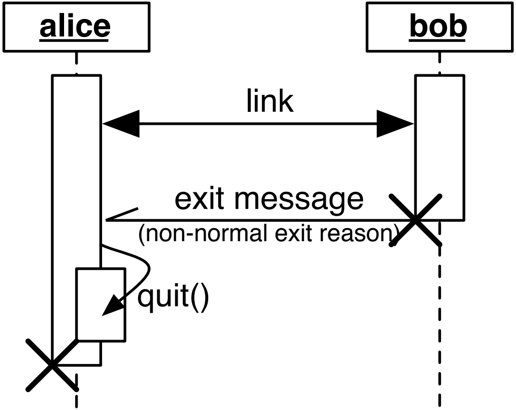
Typical actor loop

Message Processing

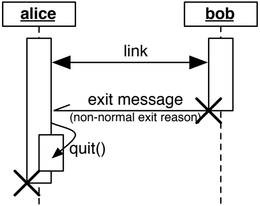


- Messages are copy-on-write tuples of any size
- Messages are buffered at the actor in a FIFO-ordered *mailbox*
- Actors set a partial function f as (replaceable) message handler
- Runtime skips each message M if $f(M)$ is undefined
- Unmatched (skipped) messages remain in the actor's mailbox
- Each receive operation begins with the oldest element

Fault Tolerance – Linking Actors



Fault Tolerance – Linking Actors

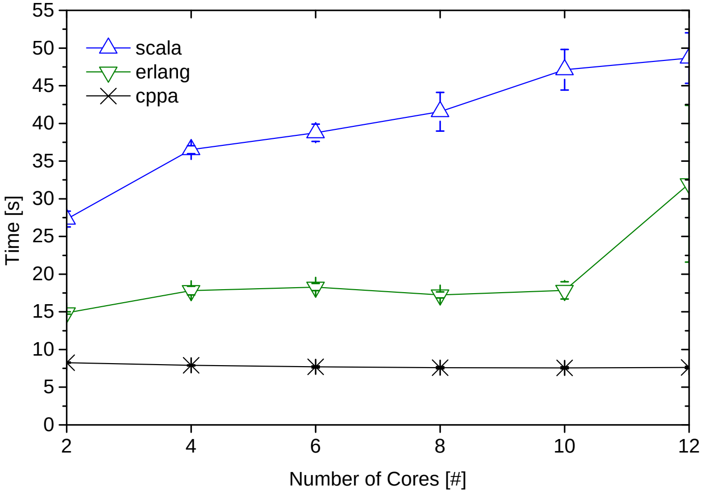


- Actors can *link* their lifetime
- Errors are propagated through exit messages
- When receiving an exit message:
  - Actors fail for the same reason per default
  - Actors can *trap* exit messages to handle failure manually
- Build systems where all actors are alive or have collectively failed

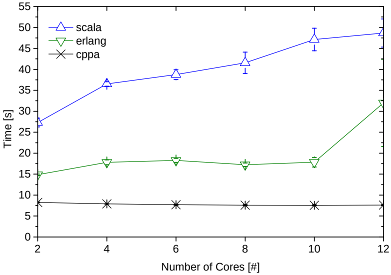
Performance for N:1 Communication

- 1 receiving actor
- 20 threads, each sending 1,000,000 messages
- Mailbox of receiving actor acts as a shared resource

Performance for N:1 Communication



Performance for N:1 Communication



- `libcppa` exhibits no concurrency penalty for up to 12 cores
- Erlang is at best 2–3 times slower than `libcppa`
- Akka's scheduling suboptimal for N:1 communication

Partial Functions in libcppa

```
partial_function f {
  on("hello") >> [] {
    cout << "hello!" << endl;
  },
  on(atom("hello")) >> [] {
    cout << "atom(hello)!" << endl;
  },
  on_arg_match >> [](int a, int b) {
    cout << a << ", " << b << endl;
  },
  on("hello", arg_match) >> [](const string& name) {
    cout << "hello " << name << "!" << endl;
  }
};

assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

Partial Functions in libcppa

```
partial_function f {
  on("hello") >> [] {
    cout << "hello!" << endl;
  },
  ... >> [
  ...llo)
  on_arg_match >> [](int a, int b) {
    cout << a << ", " << b << endl;
  },
  on("hello", arg_match) >> [](const string& name) {
    cout << "hello " << name << "!" << endl;
  }
};

assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

matches tuples with one (string) element of value "hello"

callback that should be invoked on a match; could take a string as argument

Partial Functions in libcppa

```
partial_function f {
  on("hello") >> [] {
    cout << "hello!" << endl;
  },
  on(atom("hello")) >> [] {
    cout << "atom(hello)!" << endl;
  },
                                int b) {
                               << endl;

                                [](const string& name) {
    cout << "hello " << name << "!" << endl;
  }
};

assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

atoms are constants, calculated at compile time from short strings (max 10 characters)

Partial Functions in libcppa

```
partial_function f {
  on("hello") >> [] {
    cout << "hello!" << endl;
  },
  on(atom("hello")) >> [] {
    cout << "atom(hello)!" << endl;
  },
  on_arg_match >> [](int a, int b) {
    cout << a << ", " << b << endl;
  },
                    [](const string& name) {
                    << "!" << endl;

};

assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

deduce types from callback signature ➔ match tuples with two integers

Partial Functions in `libcppa`

```
partial_function f {
  on("hello") >> [] {
    cout << "hello!" << endl;
  },
    cout << a << ", " << b << endl;
  },
  on("hello", arg_match) >> [](const string& name) {
    cout << "hello " << name << "!" << endl;
  }
};

assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

deduce second half of types from callback signature ➔ match tuples with two strings if first element is "hello"

Partial Functions in `libcppa`

```
partial_function f {
  on("hello") >> [] {
    cout << "hello!" << endl;
  },
  on(atom("hello")) >> [] {
    cout << "atom(hello)!" << endl;
  },
  on_arg_match >> [](int a, int b) {
    cout << a << ", " << b << endl;
  }
                      nst string& name) {
                      !" << endl;

};

assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

libcppa's pattern matching is defined only for `any_tuple`, because it requires runtime type information

Minimal Actor Example

```
void math_server() {
  become (
    on(atom("plus"), arg_match) >> [](int a, int b) {
      reply(atom("result"), a + b);
    }
  );
}
void math_client(actor_ptr ms) {
  sync_send(ms, atom("plus"), 40, 2).then(
    on(atom("result"), arg_match) >> [=](int result){
      cout << "40 + 2 = " << result << endl;
    }
  );
}
int main() {
  spawn(math_client, spawn(math_server));
  // ...
}
```

Minimal Actor Example

```
void math_server() {
  become (
    on(atom("plus"), arg_match) >> [](int a, int b) {
      reply(atom("result"), a + b);
}
void math_client(actor_ptr ms) {
  sync_send(ms, atom("plus"), 40, 2).then(
    on(atom("result"), arg_match) >> [=](int result){
      cout << "40 + 2 = " << result << endl;
    }
  );
}
int main() {
  spawn(math_client, spawn(math_server));
  // ...
}
```

set partial function as message handler; handler is used until replaced or actor is done

Minimal Actor Example

```
void math_server() {
  become (
    on(atom("plus"), arg_match) >> [](int a, int b) {
                                   "), a + b);



}
void math_client(actor_ptr ms) {
  sync_send(ms, atom("plus"), 40, 2).then(
    on(atom("result"), arg_match) >> [=](int result){
      cout << "40 + 2 = " << result << endl;
    }
  );
}
int main() {
  spawn(math_client, spawn(math_server));
  // ...
}
```

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

Minimal Actor Example

```
void math_server() {
  become (
    on(atom("plus"), arg_match) >> [](int a, int b) {
                                 , a + b);

}
void math_client(actor_ptr ms) {
  sync_send(ms, atom("plus"), 40, 2).then(
    on(atom("result"), arg_match) >> [=](int result){
      cout << "40 + 2 = " << result << endl;
    }
  );
}
int main() {
  spawn(math_client, spawn(math_server));
  // ...
}
```

this actor "loops" forever
(or until it is forced to quit)

Minimal Actor Example

```
void math_server() {
  become (
    o                   tch) >> [](int a, int b) {
                        a + b);
    }
  );
}
void math_client(actor_ptr ms) {
  sync_send(ms, atom("plus"), 40, 2).then(
    on(atom("result"), arg_match) >> [=](int result){
      cout << "40 + 2 = " << result << endl;
    }
  );
}
int main() {
  spawn(math_client, spawn(math_server));
  // ...
}
```

this actor sends one message and receives one messages

Minimal Actor Example

```
void math_server() {
  become (
    on(atom("plus"), arg_match) >> [](int a, int b) {
      reply(atom("result"), a + b);
    }
  );
}
void math_client(actor_ptr ms) {
  sync_send(ms, atom("plus"), 40, 2).then(
    on(atom("result"), arg_match) >> [=](int result){
                                 << result << endl;
}
int main() {
  spawn(math_client, spawn(math_server));
  // ...
}
```

usage example