libcppa
Type-safe Messaging Systems in C++

Dominik Charousset* and Thomas C. Schmidt*
dcharousset@acm.org, t.schmidt@ieee.org

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

May 2014 @C++Now2014

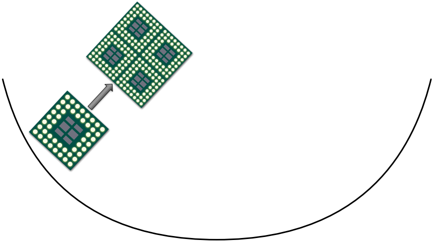
Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Agenda

Challenges of Scalable Software

Developers face not one, but multiple trends:

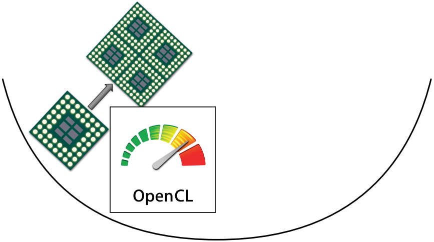
- More cores on both desktop & mobile plattforms



Challenges of Scalable Software

Developers face not one, but multiple trends:

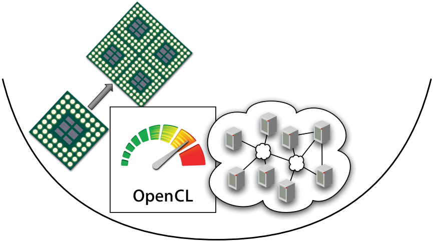
- More cores on both desktop & mobile plattforms
- SIMD components: GPUs can vastly outperform CPUs



Challenges of Scalable Software

Developers face not one, but multiple trends:

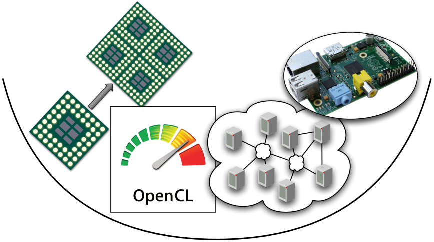
- More cores on both desktop & mobile plattforms
- SIMD components: GPUs can vastly outperform CPUs
- Cloud computing: "Infrastructure as a service"
- Internet-wide deployment



Challenges of Scalable Software

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"
- Internet-wide deployment
- Embedded HW & "The Internet of Things"

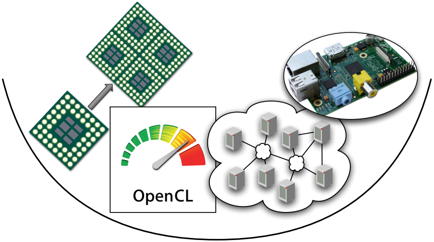


Challenges of Scalable Software

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"
- Internet-wide deployment
- Embedded HW & "The Internet of Things"

⇒ Heterogeneous platforms, concurrency & distribution



Raising the Level of Abstraction

Threads, Locks and Futures as found in the STL are **not** a sufficient abstraction. We should be enabled to ...

- Easily split application logic into as many tasks as needed
- Avoid race conditions by design (no locks!)
- Compose large systems out of small components *easily*
- Keep interfaces between software components stable:
  - Whether or not they run on the same host
  - Whether or not they run on specialized hardware

  $\Rightarrow$ Flexible composition

Raising the Level of Abstraction

Threads, Locks and Futures as found in the STL are **not** a sufficient abstraction. We should be enabled to ...

- Easily split application logic into as many tasks as needed
- Avoid race conditions by design (no locks!)
- Compose large systems out of small components *easily*
- Keep interfaces between software components stable:
  - Whether or not they run on the same host
  - Whether or not they run on specialized hardware

  $\Rightarrow$ Flexible composition

All of these criteria are met by the actor model.

Agenda

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction based on message passing

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction based on message passing
- Abstraction over deployment
  - Flexible & modular systems
  - Managing heterogeneous environments (but not yet on HW level)

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction based on message passing
- Abstraction over deployment
  - Flexible & modular systems
  - Managing heterogeneous environments (but not yet on HW level)
- Applies to both concurrency *and* distribution
  - Divide workload by spawning actors
  - Network-transparent messaging

Benefits of the Actor Model

- High-level, explicit communication between SW components
  - Robust software design: No locks, no implicit sharing
  - High level of abstraction based on message passing
- Abstraction over deployment
  - Flexible & modular systems
  - Managing heterogeneous environments (but not yet on HW level)
- Applies to both concurrency *and* distribution
  - Divide workload by spawning actors
  - Network-transparent messaging
- Provides strong failure semantics
  - Hierarchical error management
  - Re-deployment at runtime

Actors and Native Programming

- Actors have not yet entered the native programming domain
  - Need to broaden range of applications
  - Deploy actors in performance-critical systems

Actors and Native Programming

- Actors have not yet entered the native programming domain
  - Need to broaden range of applications
  - Deploy actors in performance-critical systems
- Actor systems need to include heterogeneous hardware
  - Integration of specialized HW components (GPGPU)

Actors and Native Programming

- Actors have not yet entered the native programming domain
  - Need to broaden range of applications
  - Deploy actors in performance-critical systems
- Actor systems need to include heterogeneous hardware
  - Integration of specialized HW components (GPGPU)
- Actor systems not available for embedded systems
  - Why not model the "Internet of Things" as network of actors?
  - HW platform should not dictate programming model
  - Portability & code re-use for developing IoT applications

Actors in C++11

- `libcppa` is an actor system based on C++11

Actors in C++11

- `libcppa` is an actor system based on C++11
- Focus on efficiency
  - Low memory footprint
  - Fast, lock-free mailbox implementation

Actors in C++11

- `libcppa` is an actor system based on C++11
- Focus on efficiency
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Targets both low-end and high-performance computing
  - Embedded HW
  - Multi-core systems

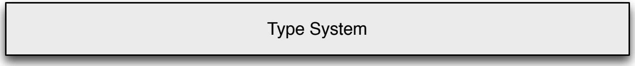
Actors in C++11

- `libcppa` is an actor system based on C++11
- Focus on efficiency
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Targets both low-end and high-performance computing
  - Embedded HW
  - Multi-core systems
- Transparent integration of OpenCL-based actors

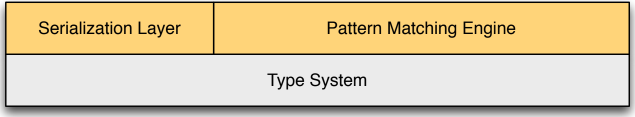
Actors in C++11

- `libcppa` is an actor system based on C++11
- Focus on efficiency
  - Low memory footprint
  - Fast, lock-free mailbox implementation
- Targets both low-end and high-performance computing
  - Embedded HW
  - Multi-core systems
- Transparent integration of OpenCL-based actors
- Uses internal DSL for pattern matching of messages

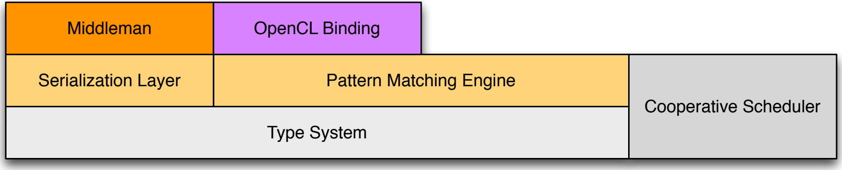
libcppa Core Architecture



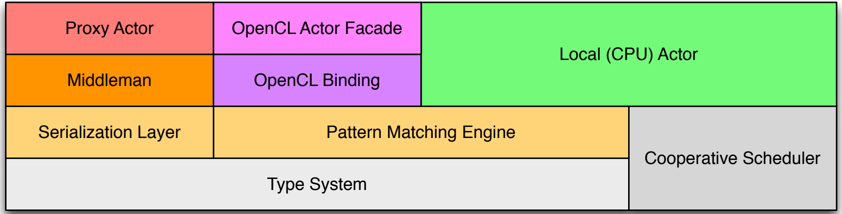
libcppa Core Architecture



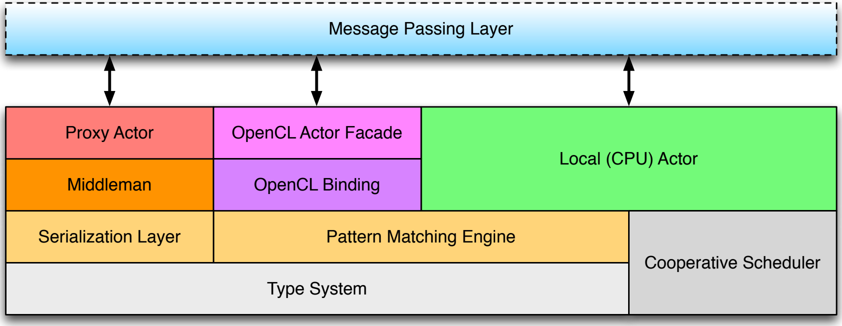
libcppa Core Architecture



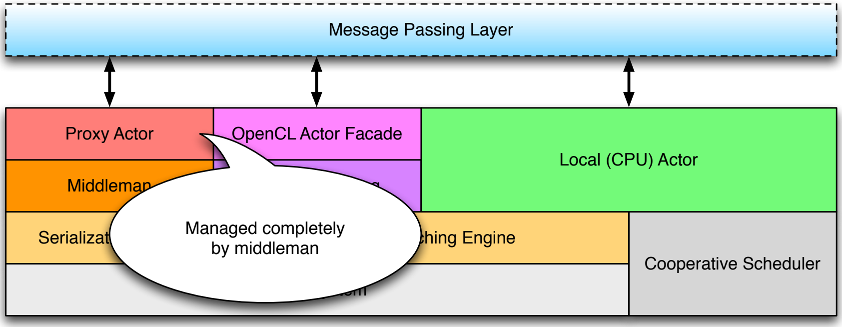
libcppa Core Architecture



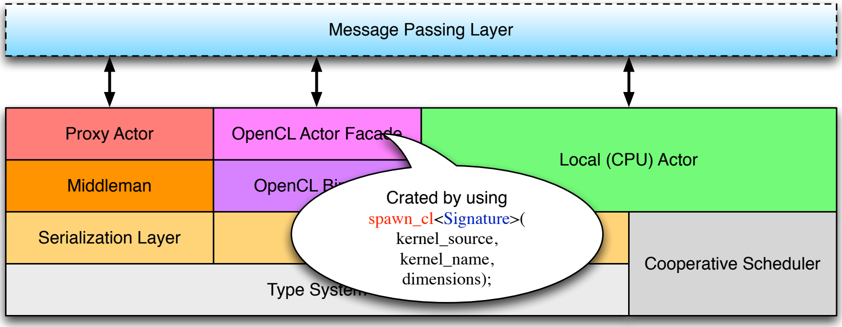
libcppa Core Architecture



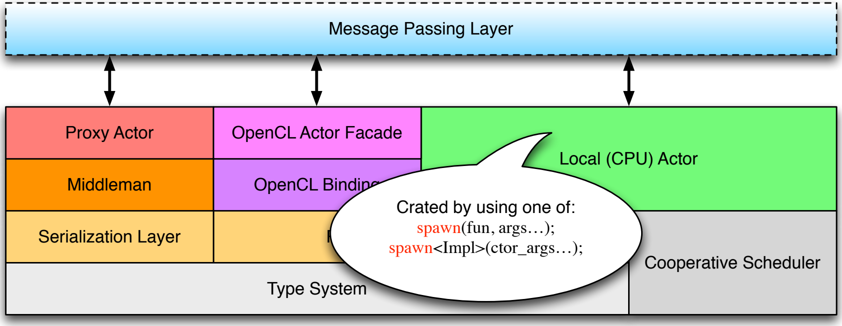
libcppa Core Architecture



libcppa Core Architecture



libcppa Core Architecture



API – Creating Actors

```
// args: constructor arguments for Impl
template<class Impl,
         spawn_options Os = no_spawn_options,
         typename... Ts>
actor spawn(Ts&&... args);

// args: functor followed by its arguments
template<spawn_options Os = no_spawn_options,
         typename... Ts>
actor spawn(Ts&&... args);
```

- Create actors from either functors or classes
- Spawn options can be used for monitoring, detaching, etc.
- Creates event-based actors per default

API – Event-based Actor Class

```
class event_based_actor : ... {

  template<typename... Ts>
  void send(actor whom, Ts&&... what);

  template<typename... Ts>
  response_handle sync_send(actor whom, Ts&&... what);

  void become(behavior bhvr);

  void quit(uint32_t reason);

  // ...

};
```

- Base for class-based actors
- Type of implicit `self` pointer for functor-based actors

API – Remote Communication

```
// makes actor accessible via network
void publish(actor whom, uint16_t port);

// get handle to remotely running actor
actor remote_actor(std::string host, uint16_t port);
```

- Message passing is network transparent
- Both local and remote actors use handles of type `actor`
- Network primitives not exposed to programmer

Example

```
behavior math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  };
}
void math_client(event_based_actor* self, actor ms) {
  self->sync_send(ms, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
      }
  );
}
// spawn(math_client, spawn(math_server));
```

Example

```
behavior math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  };
}
voi                                        self, actor ms) {
  se

      cout << "40 + 2 = " << result << endl;
      }
  );
}
// spawn(math_client, spawn(math_server));
```

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

Example

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

```
behavior math_server() {
  return
    [](in
      ret
    }
  };
}
void math_client(event_based_actor* self, actor ms) {
  self->sync_send(ms, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// spawn(math_client, spawn(math_server));
```

Example

```
behavior math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  }
}
vo                                   * self, actor ms) {
  s
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// spawn(math_client, spawn(math_server));
```

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

Example

```
behavior math_server() {
```

this actor sends one message and receives one messages

```
  };
}
void math_client(event_based_actor* self, actor ms) {
  self->sync_send(ms, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
      }
  );
}
// spawn(math_client, spawn(math_server));
```

Example

```
behavior math_server() {
  return {
    [](int a, int b) {
      return a + b;
    }
  };
}
voi                      r* self, actor ms) {
  se                     (

      cout << "40 + 2 = " << result << endl;
    }
  );
}
// spawn(math_client, spawn(math_server));
```

spawn server & client

API – Type Safety

- All functions are available as typed version
- Strongly typed actors use handles of type `typed_actor<...>`
- Interface is defined using `replies_to<...>::with<...>` notation
- Messaging to/from typed actors fully checked at compile time

API – Typed Actor Handles

Typed actor handles can be assigned to subtypes (even remote!):

```
using atype1 = typed_actor<replies_to<int>::with<int>,
                           replies_to<float>::with<float>>;
using atype2 = typed_actor<replies_to<int>::with<int>>;

atype1 a1 = spawn_typed(...);
atype2 a2 = a1; // assign to subtype
```

API – Typed Example

```
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->sync_send(ms, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// spawn(math_client, spawn_typed(math_server));
```

API – Typed Example

```
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->sync_send(ms, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// spawn(math_client, spawn_typed(math_server));
```

typedef with interface definition for convenience

API – Typed Example

```
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->sync_send(ms, 40, 2).then(
    [=](int result) {
      cout << "40 + 2 = " << result << endl;
    }
  );
}
// spawn(math_client, spawn_typed(math_server));
```

types of message handlers must match interface definition

API – Typed Example

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

messages to ms now type-checked

API – Monitoring Example

```
behavior worker(); // sometimes fails

behavior master(event_based_actor* self) {
  auto w = self->spawn<monitored>(worker);
  return {
    [=](int a, int b) {
      self->send(w, a, b);
    },
    [=](const down_msg& msg) {
      if (msg.source == w) {
        // start a new worker
        self->become(master(self));
      }
    }
  };
}
```

Agenda

Actors vs Threads

Matrix multiplication as scaling behavior showcase:

- Large number of independent tasks
- Can make use of C++11's `async`
- Simple to port algorithm to GPU (because: why not?)

Multiply Matrices – Matrix Class

```
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) {
      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_actors_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) {
  using fvec = vector<float>;
  using cfvec = const fvec&;
                      // function signature
  auto worker = spawn_cl<fvec (cfvec, cfvec)>(
                  // code, kernel name & dimensions
                  source, "multiply",
                  {matrix_size, matrix_size});
  scoped_actor self;
  self->send(worker, lhs.data(), rhs.data());
  matrix result;
  self->receive([&](fvec& res_vec) {
    result = move(res_vec);
  });
  return move(result);
}
```

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.8, 1000x1000 matrices

```
time ./simple_multiply
0m9.029s
```

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.8, 1000x1000 matrices

```
time ./simple_multiply
0m9.029s
```

```
time ./actor_multiply
0m1.164s
```

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.8, 1000x1000 matrices

```
time ./simple_multiply
0m9.029s
```

```
time ./actor_multiply
0m1.164s
```

```
time ./opencl_multiply
0m0.288s
```

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.8, 1000x1000 matrices

```
time ./simple_multiply
0m9.029s

time ./actor_multiply
0m1.164s

time ./opencl_multiply
0m0.288s

time ./async_multiply
terminate called after throwing an instance of 'std::system_error'
  what():  Resource temporarily unavailable
```

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.8, 1000x1000 matrices

```
time ./simple_multiply
0m9.029s
```

```
time ./actor_multiply
0m1.164s
```

```
time ./opencl_multiply
0m0.288s
```

```
time ./async_multiply
terminate called after throwing an instance of 'std::system_error'
  what():  Resource temporarily unavailable
```

... apparently, `std::async` is syntactic sugar for starting threads

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.8, 1000x1000 matrices

```
time ./simple_multiply
0m9.029s
```

```
time ./actor_multiply
0m1.164s
```

```
time ./opencl_multiply
0m0.288s
```

```
time ./async_multiply
terminate called after throwing an instance of 'std::system_error'
  what():  Resource temporarily unavailable
```

... apparently, `std::async` is syntactic sugar for starting threads
... and one cannot start 1,000,000 threads

Multiply Matrices – Summary

- Threads do **not** scale up to large numbers, actors do
- Spawning actors is fast
  - A million actors in $\leq 1.1\,\mathrm{s}$
  - Approach ideal speedup despite spawning $> 80\mathrm{k}$ actors per CPU
- Yes, porting algorithms to GPUs is indeed worthwhile
  - Speedup is ludicrous
  - Shouldn't surprise anybody

Agenda

Measurements

Benchmarks are based on the following implementations:

`libcppa` C++ (GCC 4.8.1) with libcppa

`scala` Scala 2.10 with the Akka library

`erlang` Erlang 5.10.2

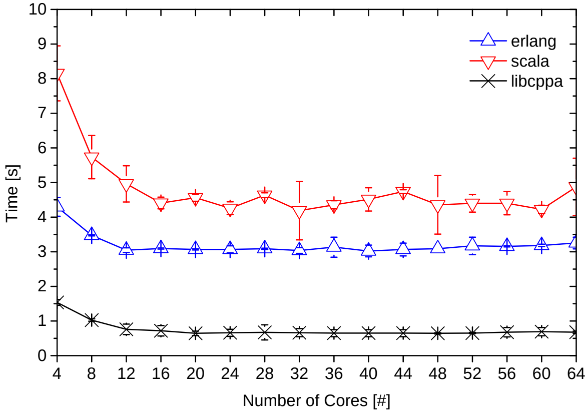
System setup:

- Four 16-core AMD Opteron 2299 MHz
- JVM configured with a maximum of 10 GB of RAM
- We vary the number of CPU cores from 4 to 64

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

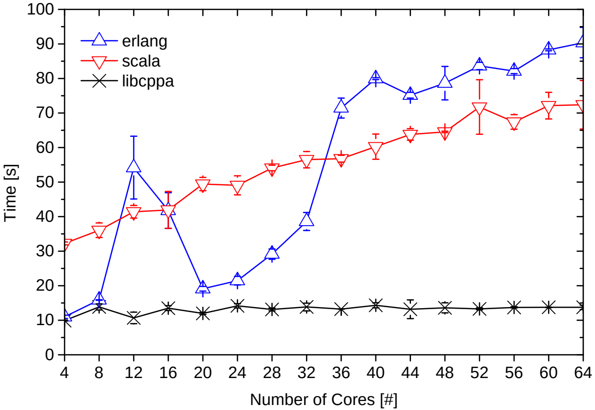


- All three implementations scale up to large actor systems
- `libcppa` performs best: 1M actors in $\leq 1\,\mathrm{s}$ for 8 or more cores

Performance of 1:N Communication

- 100 senders transmitting 100k messages each to a single receiver
- Stresses performance of receive for central actors
- More HW concurrency adds more collisions on receiver mailbox

Performance of N:1 Communication

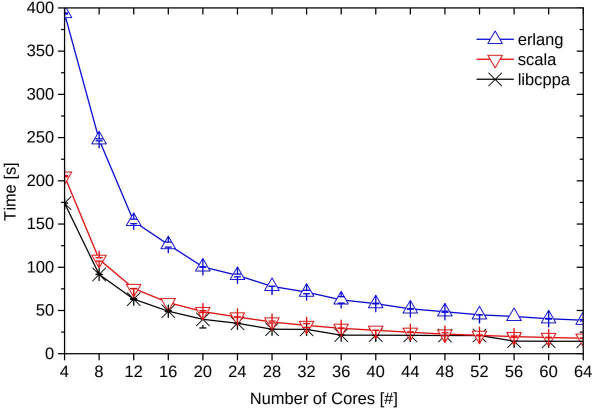


- Runtime increases significantly for Erlang and Scala
- `libcppa` remains almost constant

Performance in a Mixed Scenario

- Mixed operations under work load
- 100 rings of 50 actors each
- Token-forwarding on each ring until 1k iterations are reached
- 5 re-creations per ring
- One prime factorization per (re)-created ring to add work load
- Doubling the number of cores should (nearly) halve the runtime

Performance in a Mixed Scenario

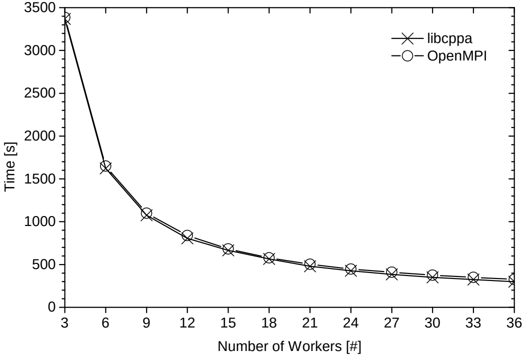


- Tail-recursive prime factorization in Scala as fast as in C++
- `libcppa` on 64 cores still $\approx 20\,\%$ faster

Scaling Behavior of Message Passing

- Calculate images of the Mandelbrot set in C++
- Distributed using (1) `libcppa` and (2) OpenMPI
  - Same source code for calculation
  - Only the message passing layers differ

Scaling Behavior of Message Passing



- Both implementations exhibit equal scaling behavior
- Doubling the number of worker nodes halves the runtime
- `libcppa` 20–30 s faster, despite higher level of abstraction

Agenda

Conclusion

- `libcppa` performs uniformly well

Conclusion

- `libcppa` performs uniformly well
- Broaden range of applications of the actor model
  - Support for GPGPU programming by integrating OpenCL
  - Small memory footprint & efficient program execution

Conclusion

- `libcppa` performs uniformly well
- Broaden range of applications of the actor model
  - Support for GPGPU programming by integrating OpenCL
  - Small memory footprint & efficient program execution
- Native C++ actor system
  - Pattern Matching for messages as internal DSL
  - High level of abstraction without sacrificing performance

Conclusion

- `libcppa` performs uniformly well
- Broaden range of applications of the actor model
  - Support for GPGPU programming by integrating OpenCL
  - Small memory footprint & efficient program execution
- Native C++ actor system
  - Pattern Matching for messages as internal DSL
  - High level of abstraction without sacrificing performance
- Currently ported to RIOT-os[1] for embedded HW support

[1] http://www.riot-os.org/

libcppa Facts Sheet

- Open source (GPLv2) `C++11` actor library
- Runs on GCC $\geq$ 4.7, Clang $\geq$ 3.2 (Linux + Mac)
- Will run on MSVC once it is C++11 complete (runs on MinGW)
- Hosted on GitHub
- Feedback & contributions always welcome!
- Hot topics in the iNET group:
    - Actors on ARM / embedded systems
    - Actors & publish/subscribe (multicast)
    - Message routing & composability

`libcppa` Facts Sheet

- Open source (GPLv2) `C++11` actor library
- Runs on GCC $\geq$ 4.7, Clang $\geq$ 3.2 (Linux + Mac)
- Will run on MSVC once it is C++11 complete (runs on MinGW)
- Hosted on GitHub
- Feedback & contributions always welcome!
- Hot topics in the iNET group:
  - Actors on ARM / embedded systems
  - Actors & publish/subscribe (multicast)
  - Message routing & composability
- Currently in (preliminary) submission process to Boost!

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