`libcppa`
An Actor Semantic for C++11

Dominik Charousset
dcharousset@acm.org

iNET RG, Department Informatik
Hamburg University of Applied Sciences

July 2013

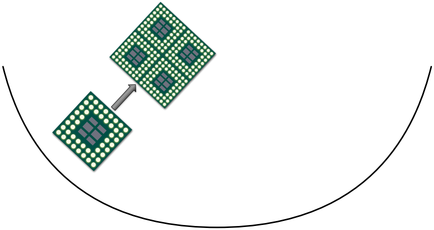
Agenda

1. Challenges of Modern Systems
2. The Problem with Implicit Sharing
   - Locks are not Composable
   - The "Right" Level of Abstraction
3. The Actor Model
   - Benefits of the Actor Model
   - `libcppa` – A `C++11` Actor Library
   - Multiply Matrices using Actors vs. `std::async`
   - Message Processing
4. `libcppa` Facts Sheet

Challenges of Modern Systems

Developers face not one, but multiple trends:

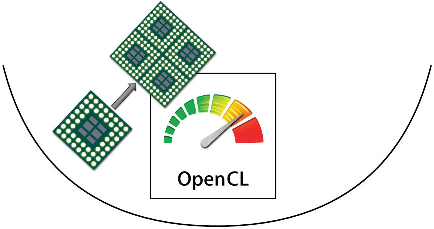
- More cores on both desktop & mobile plattforms



Challenges of Modern Systems

Developers face not one, but multiple trends:

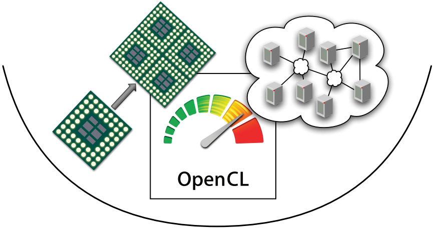
- More cores on both desktop & mobile plattforms
- GPGPU programming: GPUs can vastly outperform CPUs



Challenges of Modern Systems

Developers face not one, but multiple trends:

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

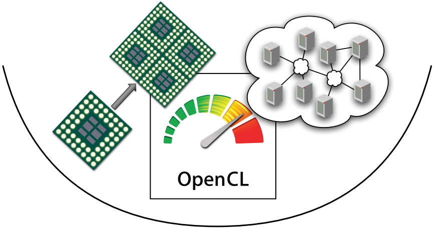


Challenges of Modern Systems

Developers face not one, but multiple trends:

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

⇒ Parallelization, specialization & distribution



Agenda

1. Challenges of Modern Systems
2. The Problem with Implicit Sharing
   - Locks are not Composable
   - The "Right" Level of Abstraction
3. The Actor Model
   - Benefits of the Actor Model
   - `libcppa` – A `C++11` Actor Library
   - Multiply Matrices using Actors vs. `std::async`
   - Message Processing
4. `libcppa` Facts Sheet

The Problem with Implicit Sharing

When writing concurrent programs:

- Stateful objects need to be synchronized (if shared)
- Developer is responsible for thread-safety
- Challenges are ...
  - Race conditions (“solved” by locks)
  - Deadlocks/Lifelocks (caused by locks)
  - Poor scalability due to queueing (Coarse-Grained Locking)
  - Very high complexity (Fine-Grained Locking)
- Time-dependent errors make testing (almost) impossible

The Problem with Implicit Sharing

When writing concurrent programs:

- Stateful objects need to be synchronized (if shared)
- Developer is responsible for thread-safety
- Challenges are ...
  - Race conditions (“solved” by locks)
  - Deadlocks/Lifelocks (caused by locks)
  - Poor scalability due to queueing (Coarse-Grained Locking)
  - Very high complexity (Fine-Grained Locking)
- Time-dependent errors make testing (almost) impossible

⇒ Expert knowledge & experience required

Locks are not Composable

"Mutable, stateful objects are the new spaghetti code."
– Rich Hickey

Locks are not Composable

“Mutable, stateful objects are the new spaghetti code.”
– Rich Hickey

- Libraries with threads & locks are no longer black boxes
- Composition of two thread-safe classes not necessarily thread-safe
- User has to know about implementation details:
  - Which code runs asynchronously/where?
  - Which functions are “thread-safe”?
  - Which function uses which lock?

Locks are not Composable

"Mutable, stateful objects are the new spaghetti code."
– Rich Hickey

- Libraries with threads & locks are no longer black boxes
- Composition of two thread-safe classes not necessarily thread-safe
- User has to know about implementation details:
  - Which code runs asynchronously/where?
  - Which functions are "thread-safe"?
  - Which function uses which lock?

⇒ Wrong level of abstraction

The "Right" Level of Abstraction

A programming paradigm should enable us to ...

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

  ⇒ Flexible composition

Agenda

1. Challenges of Modern Systems

2. The Problem with Implicit Sharing
   - Locks are not Composable
   - The “Right” Level of Abstraction

3. The Actor Model
   - Benefits of the Actor Model
   - `libcppa` – A `C++11` Actor Library
   - Multiply Matrices using Actors vs. `std::async`
   - Message Processing

4. `libcppa` Facts Sheet

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

Benefits of the Actor Model

- High-level, explicit communication: no locks, no implicit sharing
- A lightweight implementation allows millions of active actors
- Applies to both concurrency *and* distribution
  - Divide workload by spawning actors
  - Network-transparent messaging

libcppa – A C++11 Actor Library

libcppa provides an actor semantic for C++11

- Raises the level of abstraction (ease of development)
- Implements lightweight actors (ease of concurrency)
- Offers transparent OpenCL layer (ease of composition)
- Operates network transparent (ease of distribution)

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 >> [&](vector<float>& vec) {
    result = move(vec);
  });
  return result;
}
```

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

```
time ./actor_multiply
0m2.428s
```

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

```
time ./actor_multiply
0m2.428s
```

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

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

time ./actor_multiply
0m2.428s

time ./opencl_multiply
0m0.288s

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

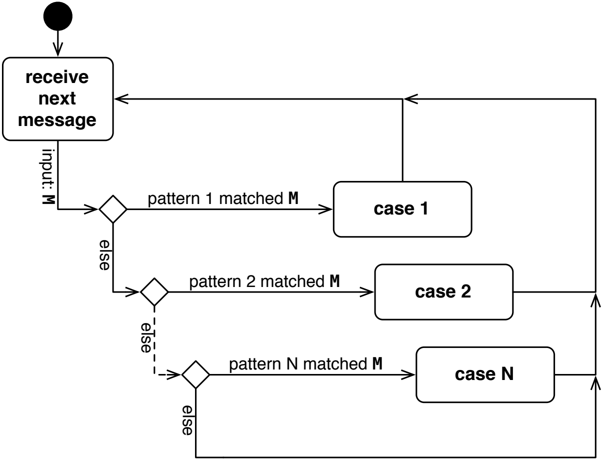
Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

time ./actor_multiply
0m2.428s

time ./opencl_multiply
0m0.288s

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

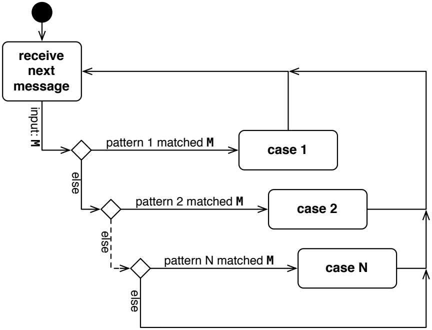
… apparently, one cannot start 1,000,000 threads

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

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;

  on("hello", arg_match) >> [](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) {
                               e << "!" << endl;

};
```

deduce types from callback signature ➔ match tuples with two integers

```
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;
  },
  on(atom("hello")) >> [] {
                                 l;

    _                 []             ) {
    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));
  // ...
}
```

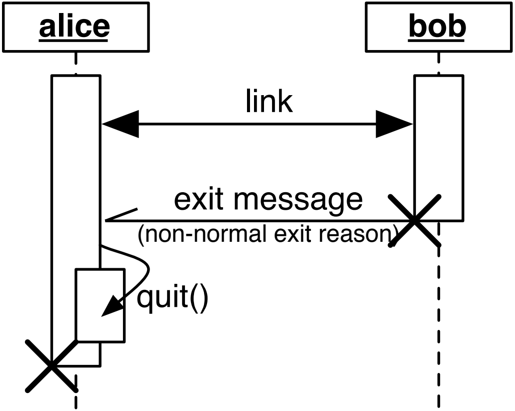
this actor sends one message and receives one messages

Minimal Actor Example

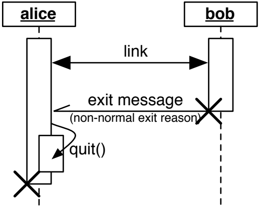
```
void math_server() {
  become (
    on(atom("plus"), arg_match) >> [](int a, int b) {
      reply(atom("result"), a + b);
    }
  );
}
void math_client(actor_ptr ms) {
  sync_send(ms, atom("plus"), 40, 2).then(
    on(atom("result"), arg_match) >> [=](int result){
                             " 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



}
int main() {
  spawn(math_client, spawn(math_server));
  // ...
}
```

usage example

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

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
  auto bob = spawn<linked>(bob_fun);
  send(bob, "hello bob");
  become ( /* will bob ever call back? */ );
}
void carl() {
  self->trap_exit(true);
  auto alice = spawn<linked>(alice_fun);
  become (
    on(atom("EXIT"), arg_match) >> [](uint32_t r) {
      if (r != exit_reason::normal)
        cout << "something went wrong..." << endl;
    }
  );
}
```

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
  auto bob = spawn<linked>(bob_fun);
  send(bob, "hello bob");
  become ( /* will bob ever call back? */ );
}
vo

  auto alice = spawn<linked>(alice_fun);
  become (
    on(atom("EXIT"), arg_match) >> [](uint32_t r) {
      if (r != exit_reason::normal)
        cout << "something went wrong..." << endl;
    }
  );
}
```

spawn bob with linked lifetime:
if bob fails, alice fails as well
(and vice versa)

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
  auto bob = spawn<linked>(bob_fun);
  send(bob, "hello bob");
  become ( /* will bob ever call back? */ );
}
void carl() {
  self->trap_exit(true);
  auto alice = spawn<linked>(alice_fun);
  become (
                           tch) >> [](uint32_t r) {
                            normal)
                            ent wrong..." << endl;
    }
  );
}
```

self always points to the running actor itself

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
  auto bob = spawn<linked>(bob_fun);
  send(bob, "hello bob");
  become ( /* will bob ever call back? */ );
}
void carl() {
  self->trap_exit(true);
  auto alice = spawn<linked>(alice_fun);
  become (
                            tch) >> [](uint32_t r) {
                            normal)
                            ent wrong..." << endl;
    }
  );
}
```

receive exit messages as ordinary messages; overriding the default behavior

Linking Actors in `libcppa` – Example

```
void bob_fun(); // will fail
void alice_fun() {
  auto bob = spawn<linked>(bob_fun);
  send(bob, "hello bob");
  become ( /* will bob ever call back? */ );
}
void carl() {
  self->trap_exit(true);
  auto alice = spawn<linked>(alice_fun);
  become (
    on(atom("EXIT"), arg_match) >> [](uint32_t r) {
      i                            l)
                                  rong..." << endl;
    }
  );
}
```

carl traps exit messages of alice, alice would fail whenever carl fails (default behavior)

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
  auto bob = spawn<linked>(bob_fun);
  send(bob, "hello bob");
                                   back? */ );
}
vo
  auto alice = spawn<linked>(alice_fun);
  become (
    on(atom("EXIT"), arg_match) >> [](uint32_t r) {
      if (r != exit_reason::normal)
        cout << "something went wrong..." << endl;
    }
  );
}
```

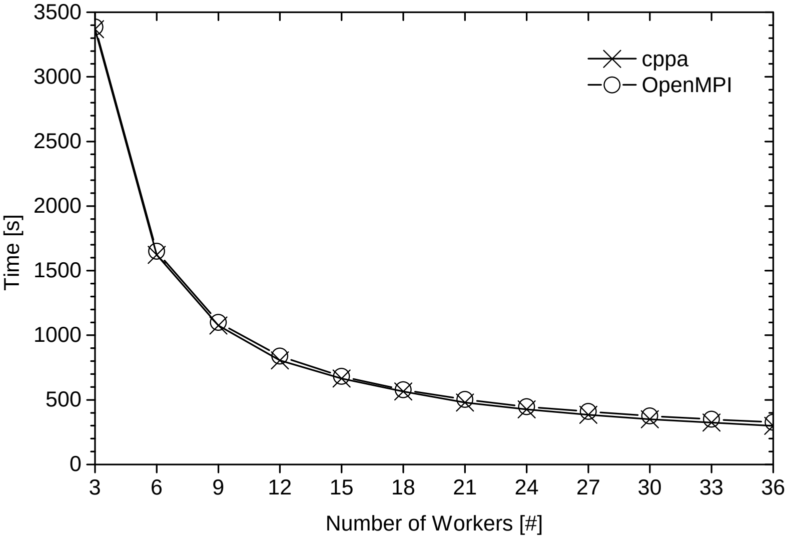
exit messages always consist of the atom 'EXIT' and the exit reason as uint32

Linking Actors in libcppa – Example

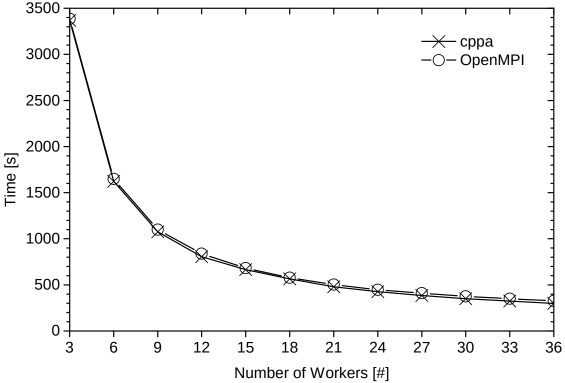
```
void bob_fun(); // will fail
void alice_fun() {
  auto bob = spawn<linked>(bob_fun);
  send(bob, "hello bob");
  become ( /* will bob ever call back? */ );
}
vo
                p       ked>(alice_fun);
  become (
    on(atom("EXIT"), arg_match) >> [](uint32_t r) {
      if (r != exit_reason::normal)
        cout << "something went wrong..." << endl;
    }
  );
}
```

a normal exit reason would indicate that alice is done (no failure occurred)

Message Processing Performance



Message Processing Performance



- Calculation of Mandelbrot set in a distributed system
- Same C++ implementation for both programs
- Despite higher level of abstraction, `libcppa` up 23 s faster

Agenda

1. Challenges of Modern Systems

2. The Problem with Implicit Sharing
   - Locks are not Composable
   - The “Right” Level of Abstraction

3. The Actor Model
   - Benefits of the Actor Model
   - `libcppa` – A `C++11` Actor Library
   - Multiply Matrices using Actors vs. `std::async`
   - Message Processing

4. `libcppa` Facts Sheet

libcppa Facts Sheet

- Open source (LGPLv2) C++11 actor library
- Runs on GCC $\geq$ 4.7, Clang $\geq$ 3.2 (Linux + Mac)
- Will run on Windows as soon as MSVC supports required features
- Hosted on GitHub
- Feedback & contributions always welcome!

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