C++ Actor Framework

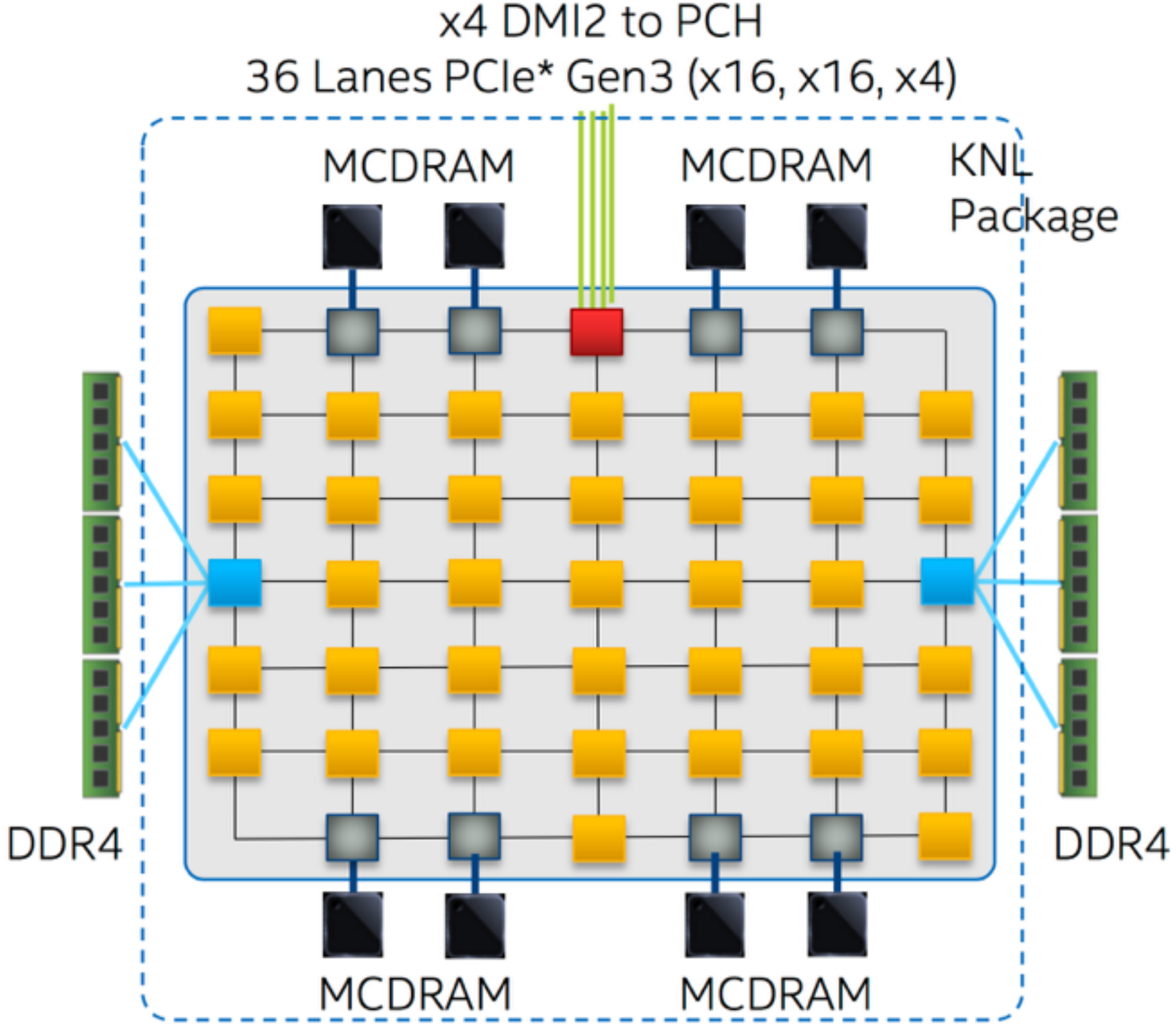
Transparent Scaling from IoT to Datacenter Apps

Matthias Vallentin

UC Berkeley

Berkeley C++ Meetup

November 2, 2016


x4 DMI2 to PCH
36 Lanes PCIe* Gen3 (x16, x16, x4)
MCDRAM
MCDRAM
KNL
Package
DDR4
DDR4
MCDRAM
MCDRAM

x4 DMI2 to PCH
36 Lanes PCIe* Gen3 (x16, x16, x4)

MCDRAM MCDRAM KNL Package

Cache Model	Flat Model	Hybrid Model

64B cache lines direct-mapped

16GB MCDRAM

DRAM

Physical Address

8GB/ 16GB MCDRAM

Up to 384 (Only 32 for coprocessor) GB DRAM

Split Options[2]: 25/75% or 50/50%

8 or 12GB MCDRAM

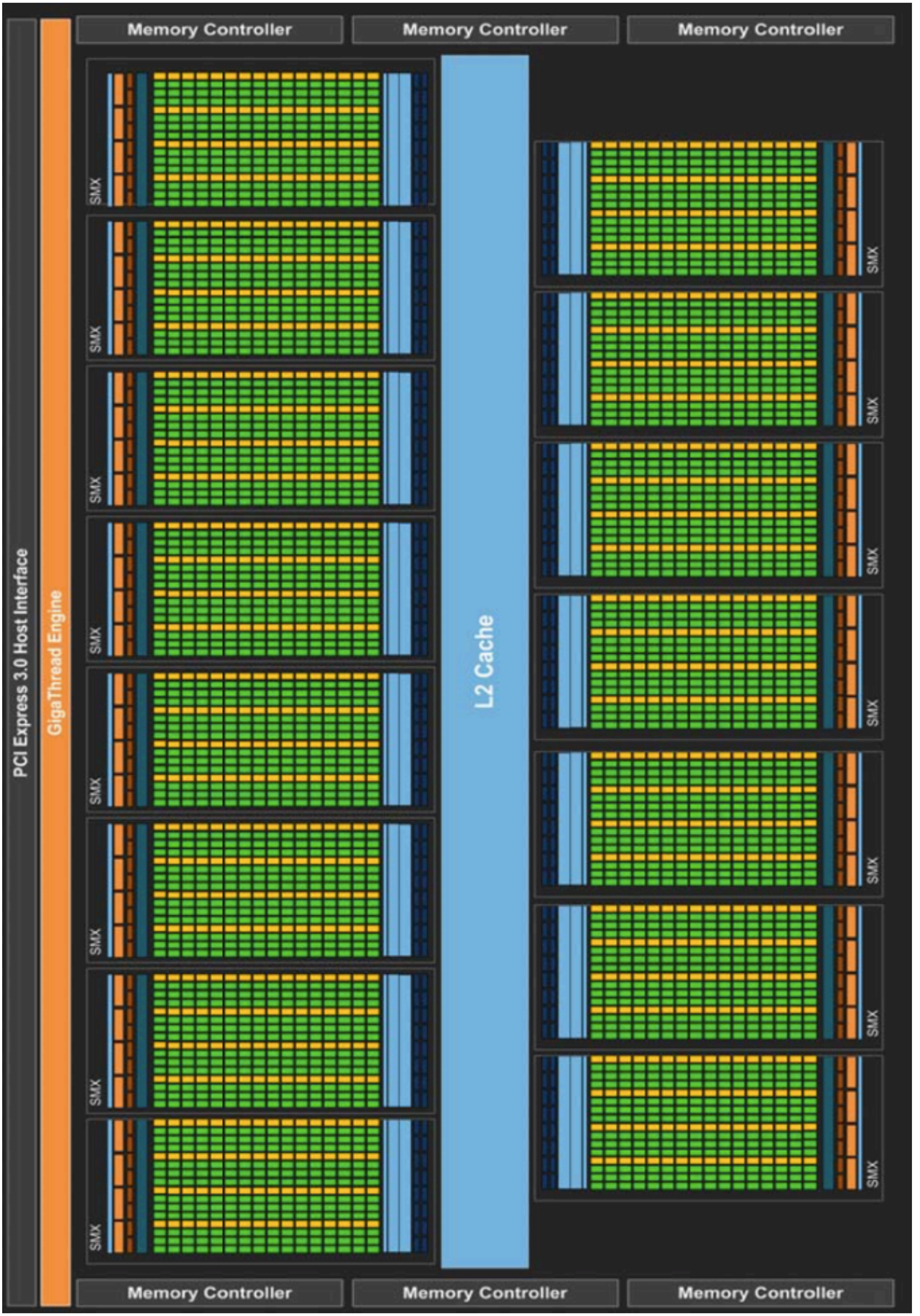
8 or 4 GB MCDRAM

DRAM

DDR4 DDR4

MCDRAM MCDRAM


TESLA


Memory Controller
Memory Controller
Memory Controller
PCI Express 3.0 Host Interface
GigaThread Engine
L2 Cache
SMX
Memory Controller
Memory Controller
Memory Controller

Memory Controller
Memory Controller
Memory Controller

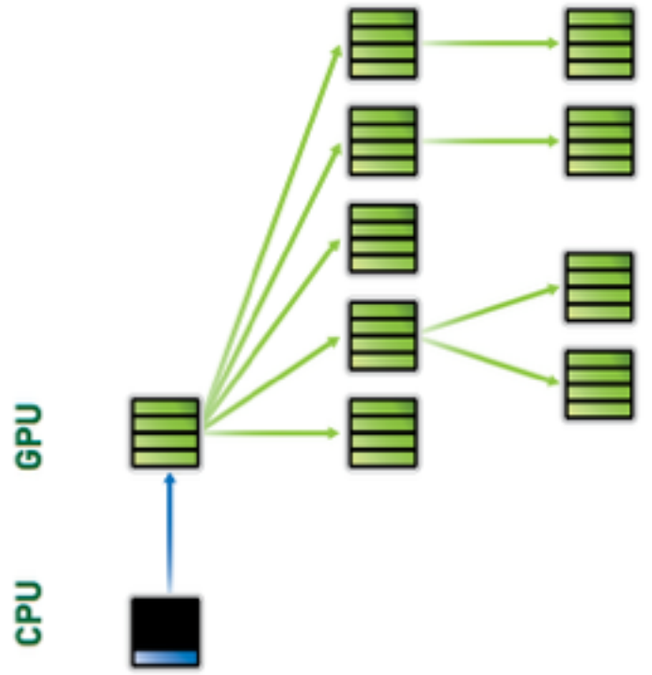
PCI Express 3.0 Host Interface

GigaThread Engine

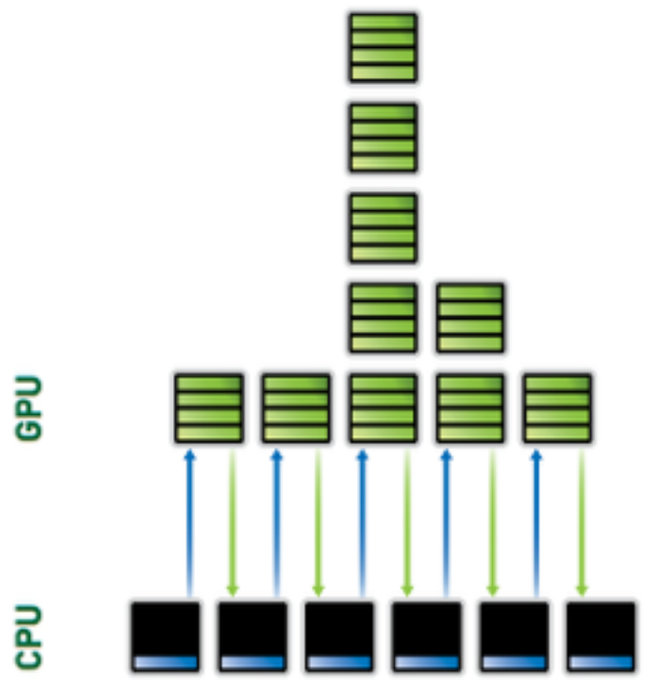
SMX

DYNAMIC PARALLELISM

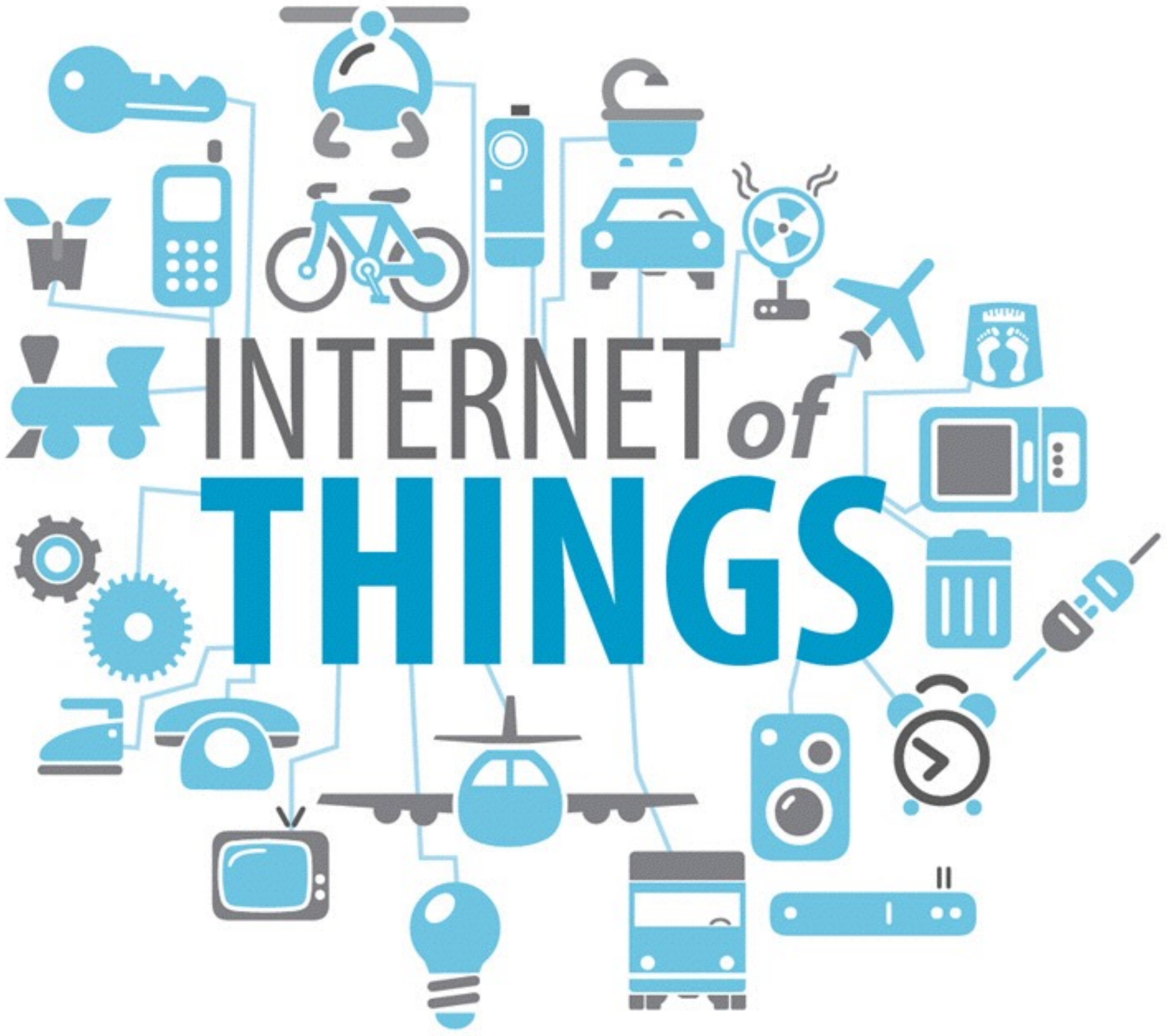
CPU
GPU



CPU
GPU

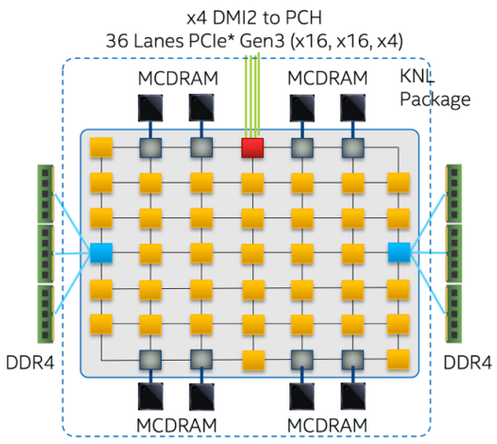


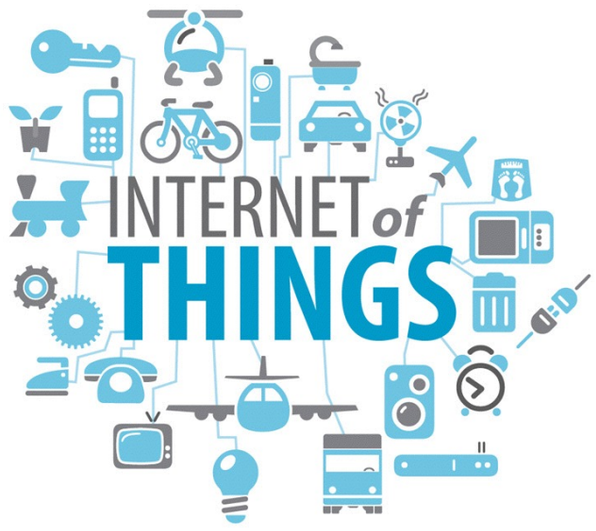
Memory Controller
Memory Controller
Memory Controller


INTERNET of
THINGS

Heterogeneity

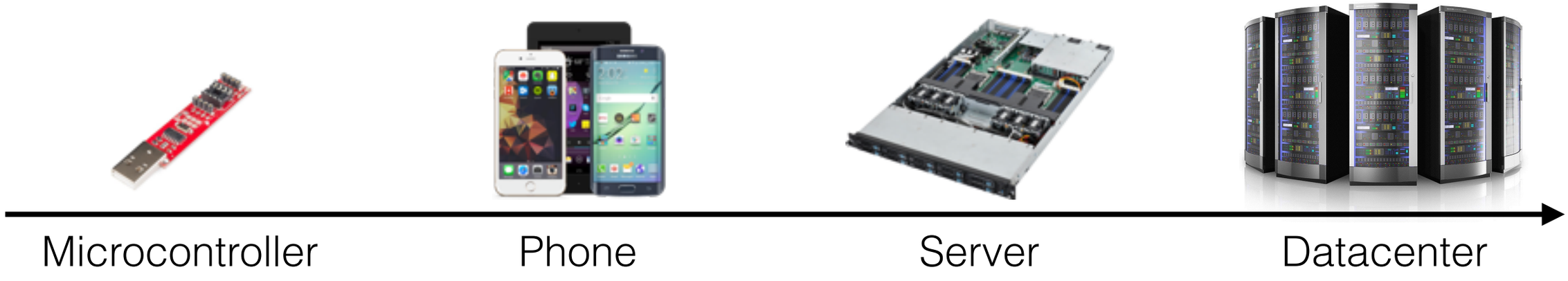
- More cores on desktops and mobile
- Complex accelerators/co-processors
- Highly distributed deployments
- Resource-constrained devices





Scalable Abstractions

- **Uniform API** for concurrency and distribution
- **Compose** small components into large systems
- **Scale** runtime from IoT to HPC



Challenges

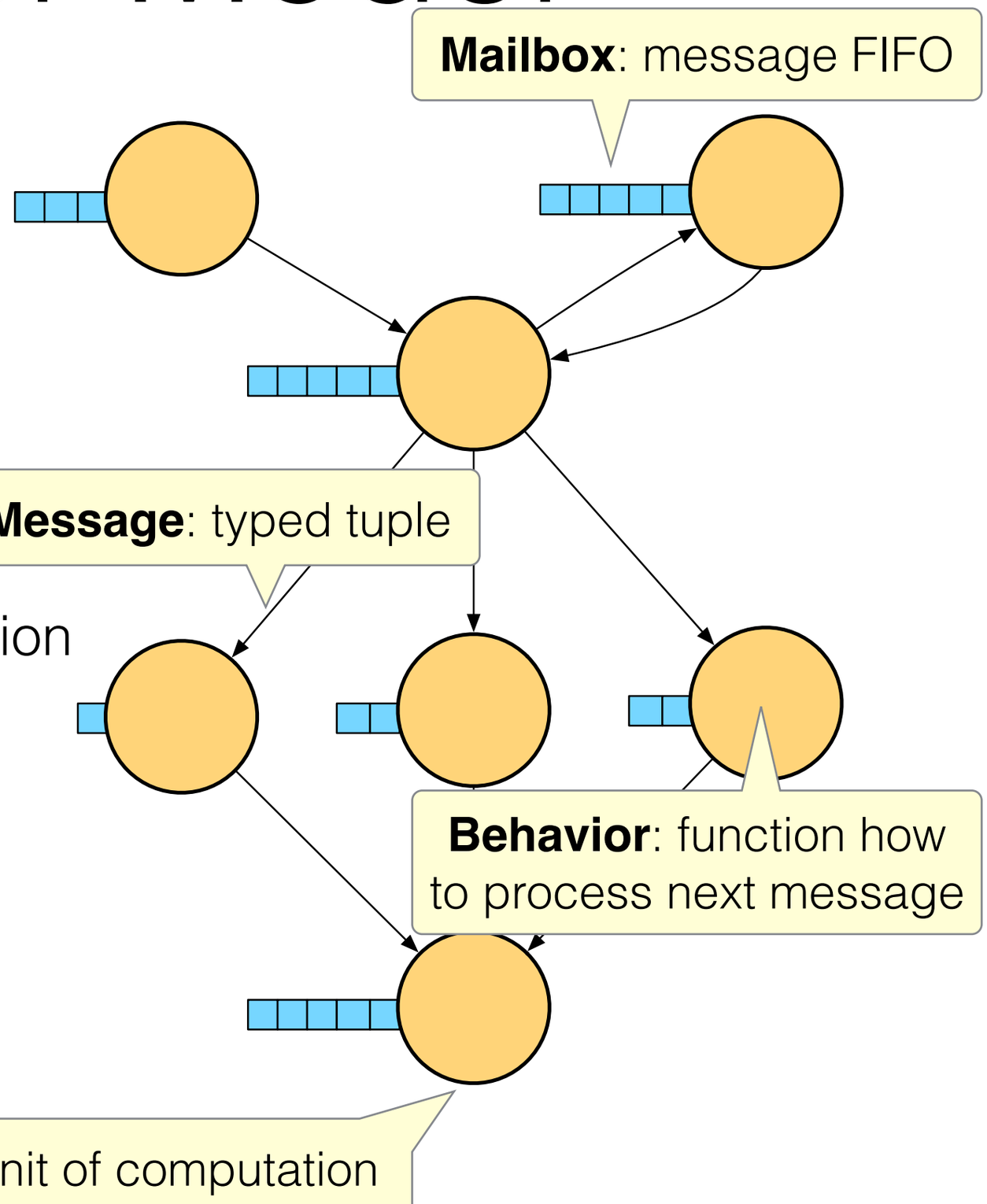
- Scalability only by **parallelizing computation**
- Writing **concurrent code** is hard
  - Race conditions
  - Livelocks, deadlocks
  - Composition
  - Non-determinism

➡ **Expert knowledge** and **experience** required

Actor Model

The Actor Model

- Devised by Hewitt in 1973
- Asynchronous message passing
- No mutable shared state
- Network-transparent communication
- Hierarchical fault propagation
- Deployment orthogonal to application logic



Actor Semantics

- All actors execute **concurrently**
- Actors are **reactive**
- In response to a message, an actor can do *any* of:
  1. Create (*spawn*) new actors
  2. Send messages to other actors
  3. Designate a behavior for the next message

CAF

(C++ Actor Framework)

Example #1

An **actor** is typically implemented as a **function**

```
behavior adder() {
  return {
    [](int x, int y) {
      return x + y;
    },
    [](double x, double y) {
      return x + y;
    }
  };
}
```

A list of **lambdas** determines the **behavior** of the actor.

A non-void return value sends a **response message** back to the sender

Example #2

```
int main() {
  actor_system_config cfg;
  actor_system sys{cfg};
  // Create (spawn) our actor.
  auto a = sys.spawn(adder);
  // Send it a message.
  scoped_actor self{sys};
  self->send(a, 40, 2);
  // Block and wait for reply.
  self->receive(
    [](int result) {
      cout << result << endl; // prints “42”
    }
  );
}
```

Encapsulates all global state (worker threads, actors, types, etc.)

Spawns an actor valid only for the current scope.

Example #2

```
int main() {
  actor_system_config cfg;
  actor_system sys{cfg};
  // Create (spawn) our actor.
  auto a = sys.spawn(adder);
  // Send it a message.
  scoped_actor self{sys};
  self->send(a, 40, 2);
  // Block and wait for reply.
  self->receive(
    [](int result) {
      cout << result << endl; // prints "42"
    }
  );
}
```

Blocking

Example #3

```
auto a = sys.spawn(adder);
sys.spawn(
  [=](event_based_actor* self) -> behavior {
    self->send(a, 40, 2);
    return {
      [=](int result) {
        cout << result << endl;
        self->quit();
      }
    };
  }
);
```

Optional first argument to running actor.

Capture by value because **spawn** returns immediately.

Designate how to handle next message.
(= set the actor behavior)

Example #3

```
auto a = sys.spawn(adder);
```

```
sys.spawn(
  [=](event_based_actor* self) -> behavior {
    self->send(a, 40, 2);
    return {
      [=](int result) {
        cout << result << endl;
        self->quit();
      }
    };
  }
);
```

Non-Blocking

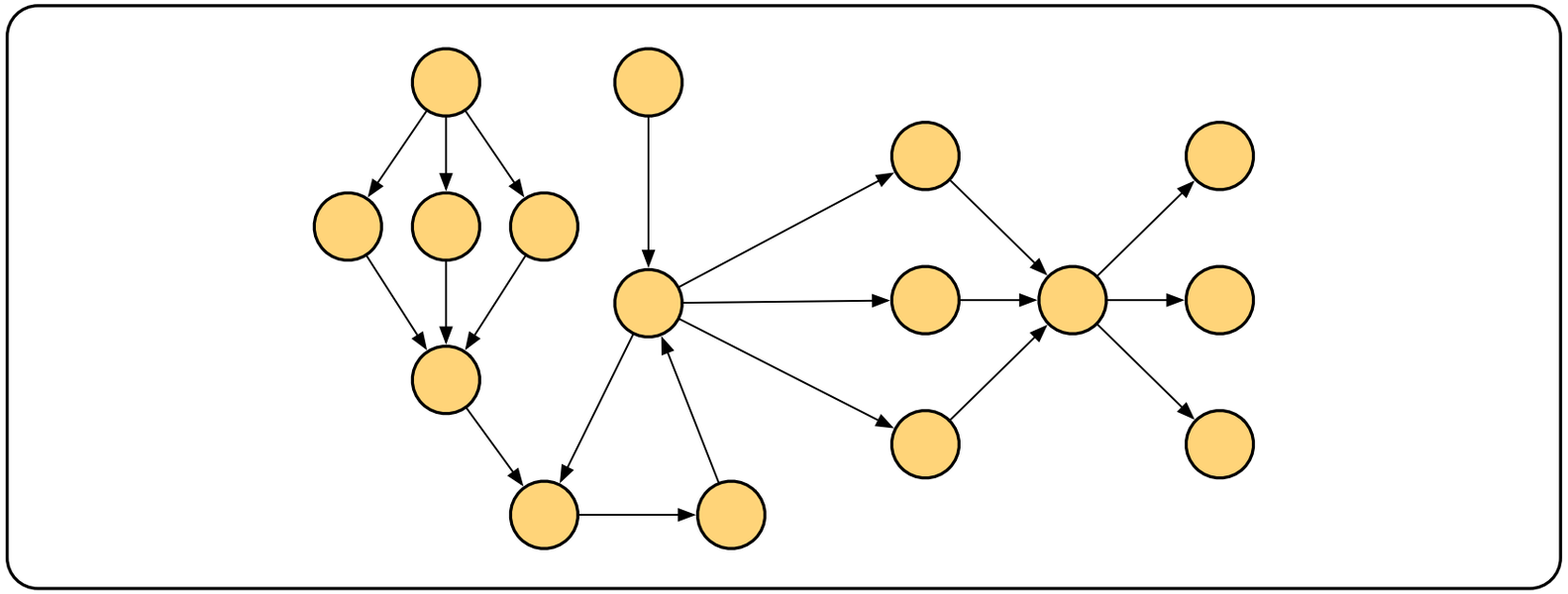
Example #4

Request-response communication requires timeout.
(`std::chrono::duration`)

```
auto a = sys.spawn(adder);
sys.spawn(
  [=](event_based_actor* self) {
    self->request(a, seconds(1), 40, 2).then(
      [=](int result) {
        cout << result << endl;
      }
    );
  }
);
```

Continuation specified as `behavior`.

Application Logic



Actor Runtime

Message Passing Abstraction

Cooperative Scheduler | Middleman / Broker | GPU Module

Operating System

Threads | Sockets | PCIe

Hardware

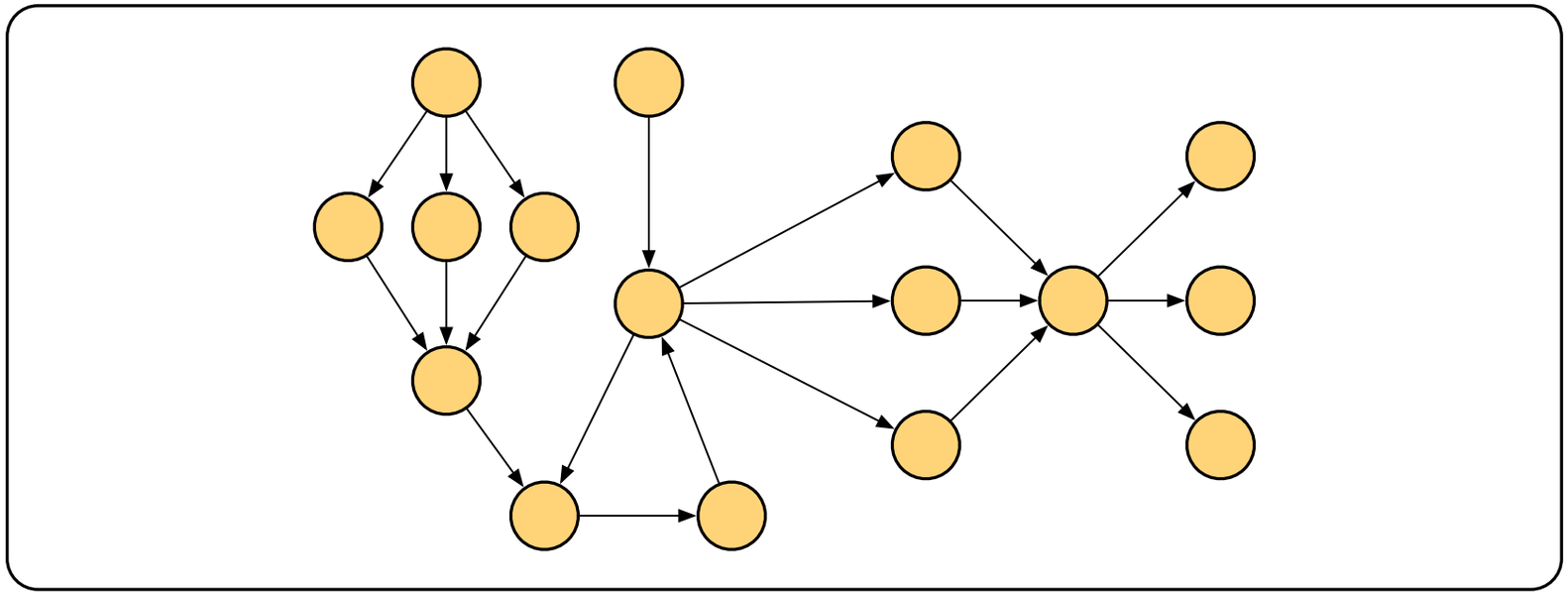
Core 0 | Core 1 | Core 2 | Core 3

L1 cache | L2 cache

Network I/O

Accelerator

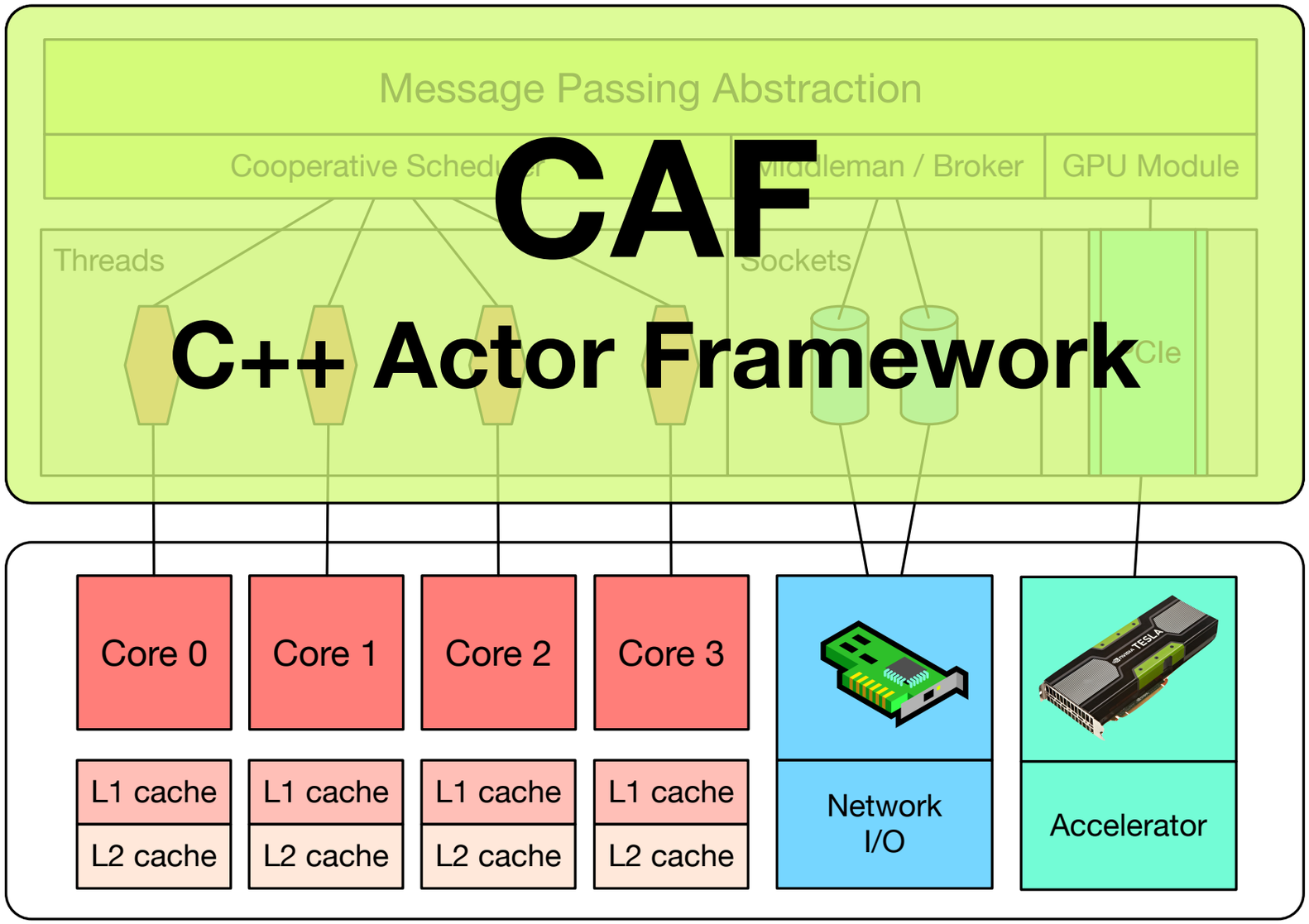
Application Logic



Actor Runtime

Operating System

Hardware

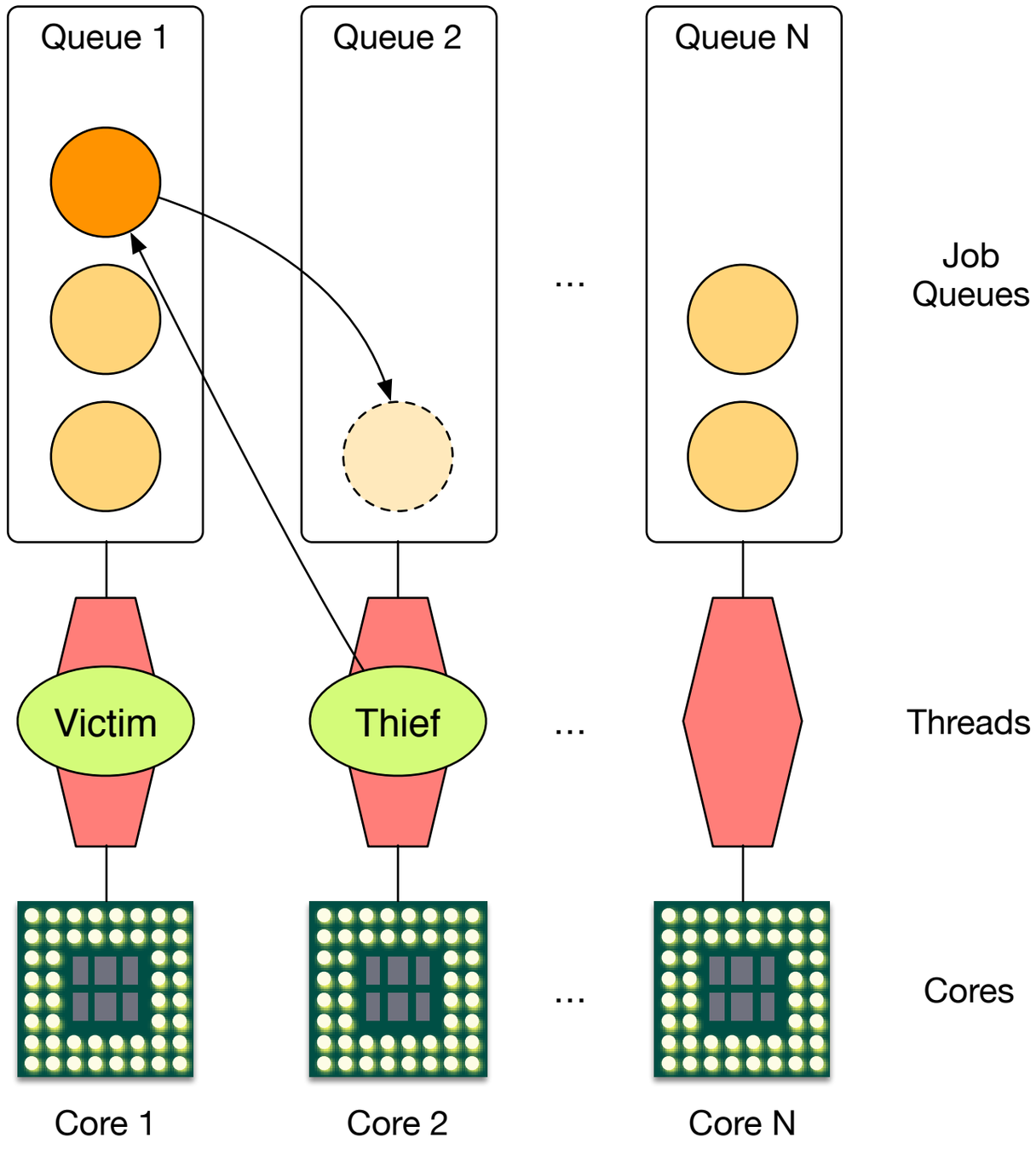


Scheduler

- Maps **N** **jobs** (= actors) to **M** **workers** (= threads)
- Limitation: **cooperative multi-tasking** in user-space
- **Issue**: actors that block
  - Can lead to **starvation** and/or scheduling imbalances
  - Not well-suited for **I/O-heavy tasks**
  - Current solution: detach "uncooperative" actors into **separate thread**

Work Stealing*

- **Decentralized**: one job queue and worker thread per core
- On empty queue, **steal** from other thread
- Efficient if stealing is a rare event
- Implementation: deque with two spinlocks



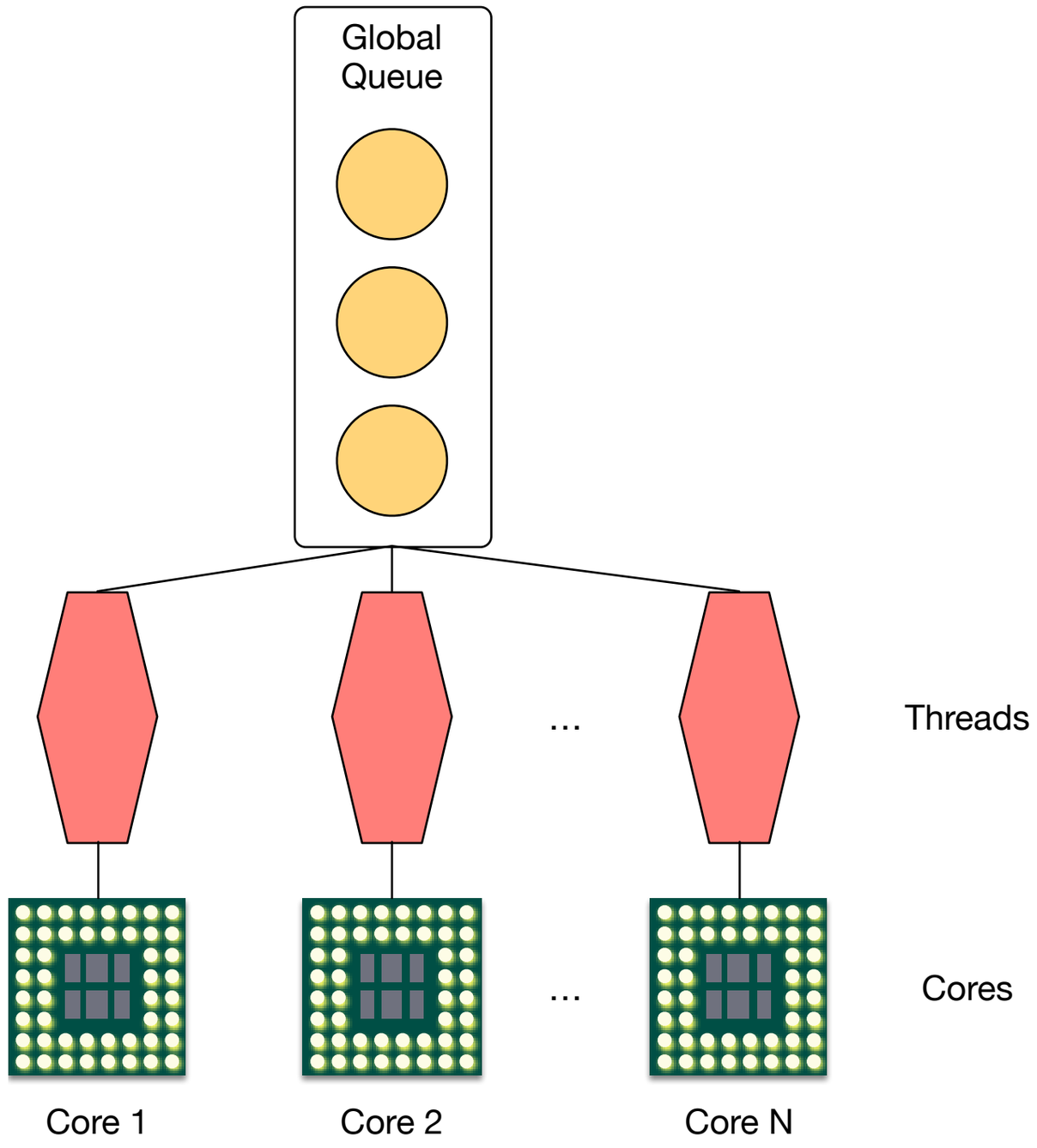
*Robert D. Blumofe and Charles E. Leiserson. *Scheduling Multithreaded Computations by Work Stealing*. J. ACM, 46(5):720–748, September 1999.

Implementation

```
template <class Worker>
resumable* dequeue(Worker* self) {
  auto& strategies = self->data().strategies;
  resumable* job = nullptr;
  for (auto& strat : strategies) {
    for (size_t i = 0; i < strat.attempts; i += strat.step_size) {
      // try to grab a job from the front of the queue
      job = self->data().queue.take_head();
      // if we have still jobs, we're good to go
      if (job)
        return job;
      // try to steal every X poll attempts
      if ((i % strat.steal_interval) == 0) {
        if (job = try_steal(self))
         return job;
      }
      if (strat.sleep_duration.count() > 0)
        std::this_thread::sleep_for(strat.sleep_duration);
    }
  }
  // unreachable, because the last strategy loops
  // until a job has been dequeued
  return nullptr;
}
```

Work Sharing

- **Centralized**: one shared global queue
- Synchronization: **mutex & CV**
- **No polling**
  - less CPU usage
  - lower throughouput
- Good **for low-power devices**
  - Embedded / IoT



Copy-On-Write

- **`caf::message`** = atomic, intrusive ref-counted tuple
  - **Immutable access** permitted
  - **Mutable access** with ref count > 1 invokes copy constructor
- **Constness deduced** from message handlers

```
auto heavy = vector<char>(1024 * 1024);
auto msg = make_message(move(heavy));
for (auto& r : receivers)
  send(r, msg);

behavior reader() {
  return {
    [=](const vector<char>& buf) {
      f(buf);
    }
  };
}

behavior writer() {
  return {
    [=](vector<char>& buf) {
      f(buf);
    }
  };
}
```

`const` access enables efficient sharing of messages

non-**`const`** access copies message contents **if ref count > 1**

- **`caf::message`** = atomic, intrusive ref-counted tuple
  - **Immutable access** permitted
  - **Mutable access** with ref count > 1 invokes copy constructor
- **Constness deduced** from message handlers
- **No data races** by design
- **Value semantics**, no complex lifetime management

```
auto heavy = vector<char>(1024 * 1024);
auto msg = make_message(move(heavy));
for (auto& r : receivers)
  send(r, msg);

behavior reader() {
  return {
    [=](const vector<char>& buf) {
      f(buf);
    }
  };
}

behavior writer() {
  return {
    [=](vector<char>& buf) {
      f(buf);
    }
  };
}
```

Type Safety

- CAF has **statically** and **dynamically typed** actors
- **Dynamic**
  - Type-erased `caf::message` hides tuple types
    - Message types checked **at runtime** only
- **Static**
  - **Type signature** verified at sender and receiver
  - Message protocol checked **at compile time**

Interface

```
// Atom: typed integer with semantics
using plus_atom = atom_constant<atom("plus")>;
using minus_atom = atom_constant<atom("minus")>;
using result_atom = atom_constant<atom("result")>;

// Actor type definition
using math_actor =
  typed_actor<
    replies_to<plus_atom, int, int>::with<result_atom, int>,
    replies_to<minus_atom, int, int>::with<result_atom, int>
  >;
```

Signature of **incoming** message

Signature of (optional) **response** message

Implementation

Dynamic

```
behavior math_fun(event_based_actor* self) {
  return {
    [](plus_atom, int a, int b) {
      return make_tuple(result_atom::value, a + b);
    },
    [](minus_atom, int a, int b) {
      return make_tuple(result_atom::value, a - b);
    }
  };
}
```

Static

```
math_actor::behavior_type typed_math_fun(math_actor::pointer self) {
  return {
    [](plus_atom, int a, int b) {
      return make_tuple(result_atom::value, a + b);
    },
    [](minus_atom, int a, int b) {
      return make_tuple(result_atom::value, a - b);
    }
  };
}
```

Error Example

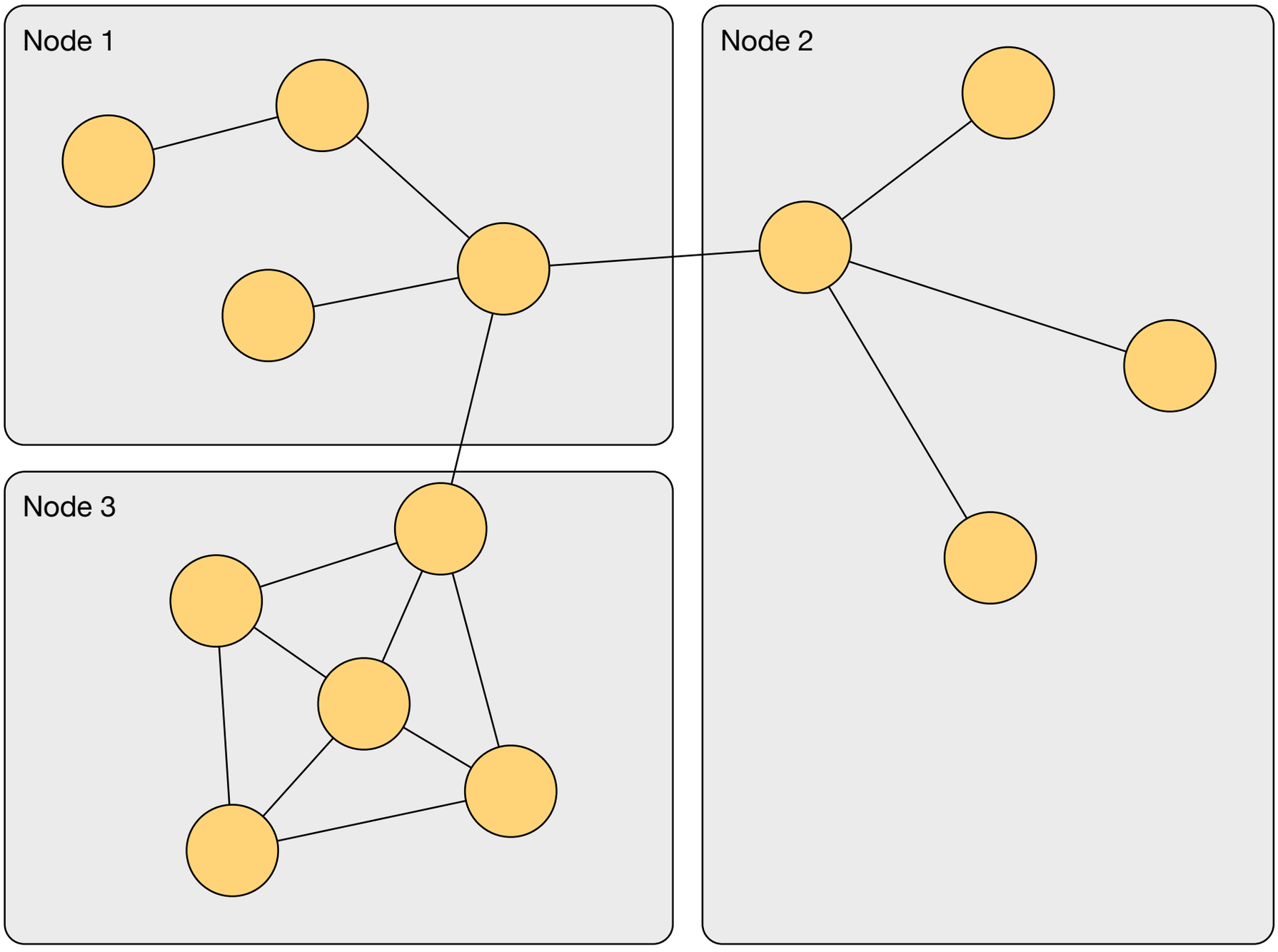
```
auto self = sys.spawn(...);
math_actor m = self->typed_spawn(typed_math);
self->request(m, seconds(1), plus_atom::value, 10, 20).then(
  [](result_atom, float result) {
    // …
  }
);
```

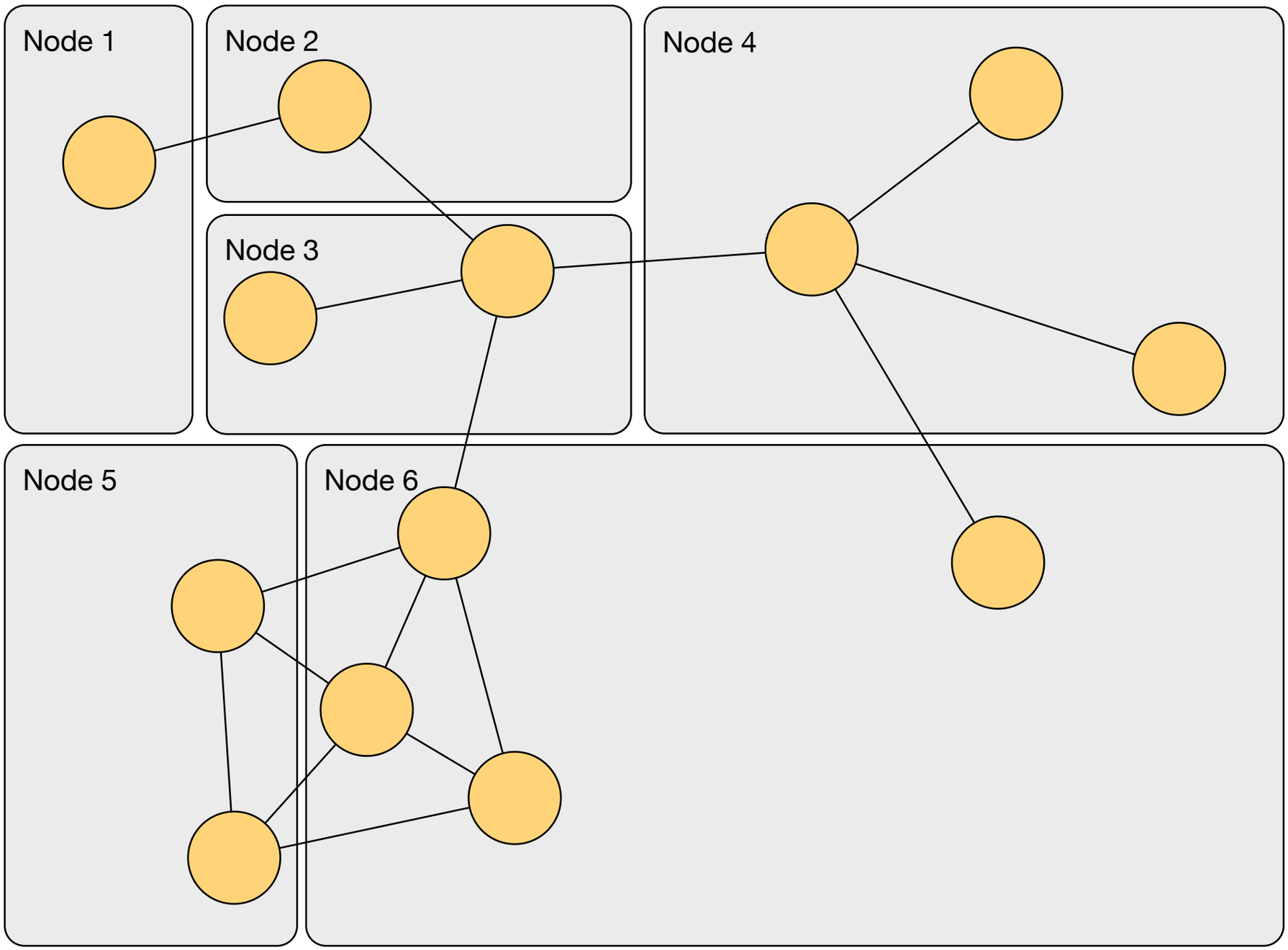
Compiler complains about invalid response type

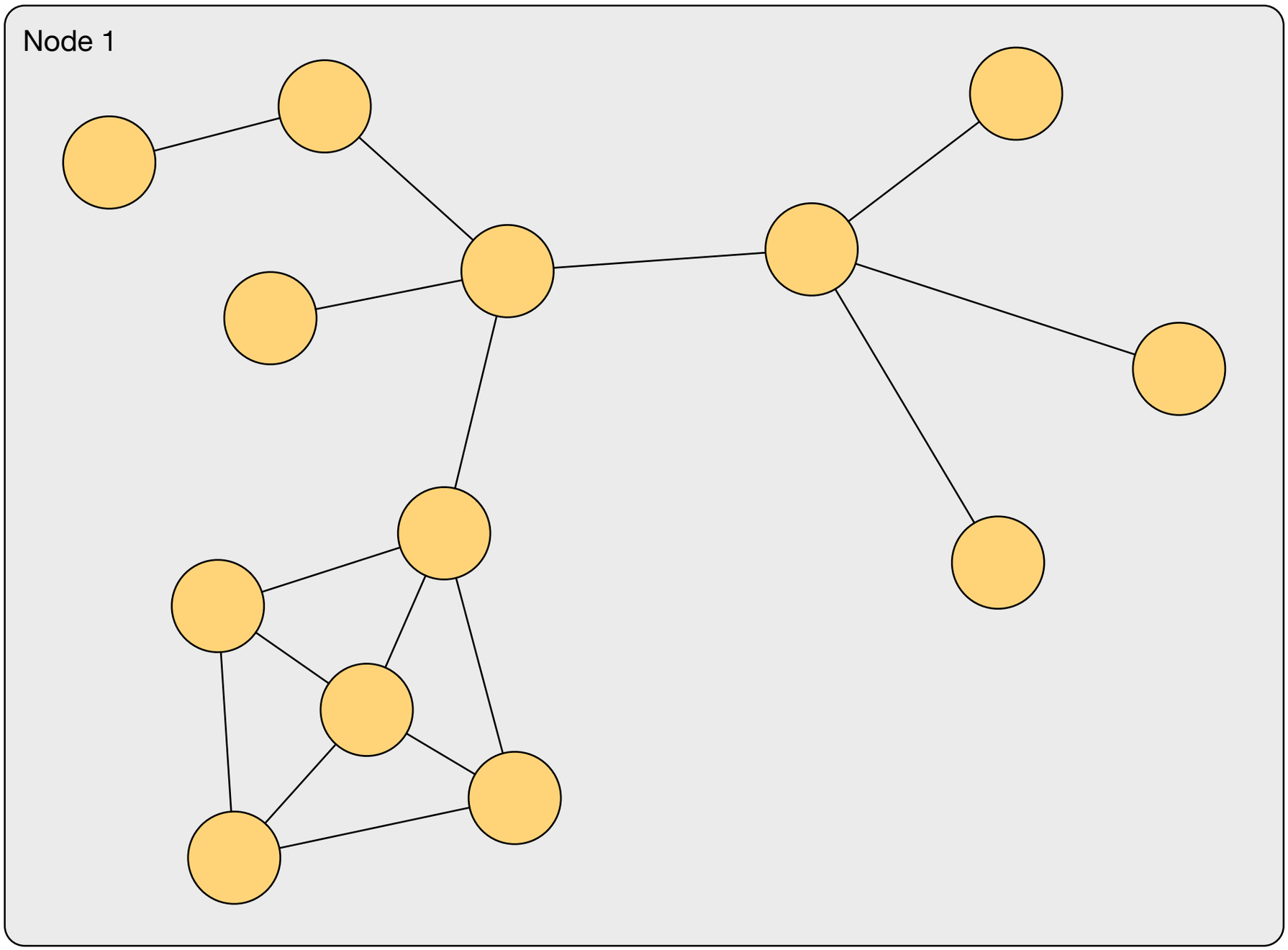
Network Transparency

Separation of **application logic** from **deployment**

- Significant **productivity gains**
  - Spend *more time* with **domain-specific code**
  - Spend *less time* with **network glue code**


Node 1
Node 2
Node 3


Node 1
Node 2
Node 3
Node 4
Node 5
Node 6


Node 1

Example

```
int main(int argc, char** argv) {
  // Defaults.
  auto host = "localhost"s;
  auto port = uint16_t{42000};
  auto server = false;
  actor_system sys{...}; // Parse command line and setup actor system.
  auto& middleman = sys.middleman();
  actor a;
  if (server) {
    a = sys.spawn(math);
    auto bound = middleman.publish(a, port);
    if (bound == 0)
      return 1;
  } else {
    auto r = middleman.remote_actor(host, port);
    if (!r)
      return 1;
    a = *r;
  }
  // Interact with actor a
}
```

Reference to CAF's network component.

Publish specific actor at a TCP port.
Returns bound port on success.

Connect to published actor at TCP endpoint.
Returns `expected<actor>`.

Failures

Components fail regularly in large-scale systems

- Actor model provides **monitors** and **links**
  - **Monitor**: subscribe to exit of actor (**unidirectional**)
  - **Link**: bind own lifetime to other actor (**bidirectional**)

Monitor Example

```
behavior adder() {
  return {
    [](int x, int y) {
      return x + y;
    }
  };
}

auto self = sys.spawn<monitored>(adder);
self->set_down_handler(
  [](const down_msg& msg) {
    cout << "actor DOWN: " << msg.reason << endl;
  }
);
```

Spawn flag denotes monitoring.
Also possible later via **`self->monitor(other);`**

Link Example

```
behavior adder() {
  return {
    [](int x, int y) {
      return x + y;
    }
  };
}

auto self = sys.spawn<linked>(adder);
self->set_exit_handler(
  [](const exit_msg& msg) {
    cout << "actor EXIT: " << msg.reason << endl;
  }
);
```

Spawn flag denotes linking.
Also possible later via **`self->link_to(other);`**

Evaluation

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

Actors vs. Threads

Matrix Multiplication

- Example for scaling computation
- Large number of independent tasks
- Can use C++11's **`std::async`**
- Simple to port to GPU

Matrix Class

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

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

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 result;
}
```

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

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 result;
}
```

Actors

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

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;
  }
)__";
```

OpenCL Actors

```
matrix opencl_multiply(const matrix& lhs,
                       const matrix& rhs) {
 auto worker = spawn_cl<float* (float* ,float*)>(
    source, "multiply", {matrix_size , matrix_size});
  actor_system_config cfg;
  actor_system sys{cfg};
  scoped_actor self{sys};
  self->send(worker, lhs.data(), rhs.data());
  matrix result;
  self->receive([&](vector<float>& xs) {
    result = move(xs);
  });
  return result;
}
```

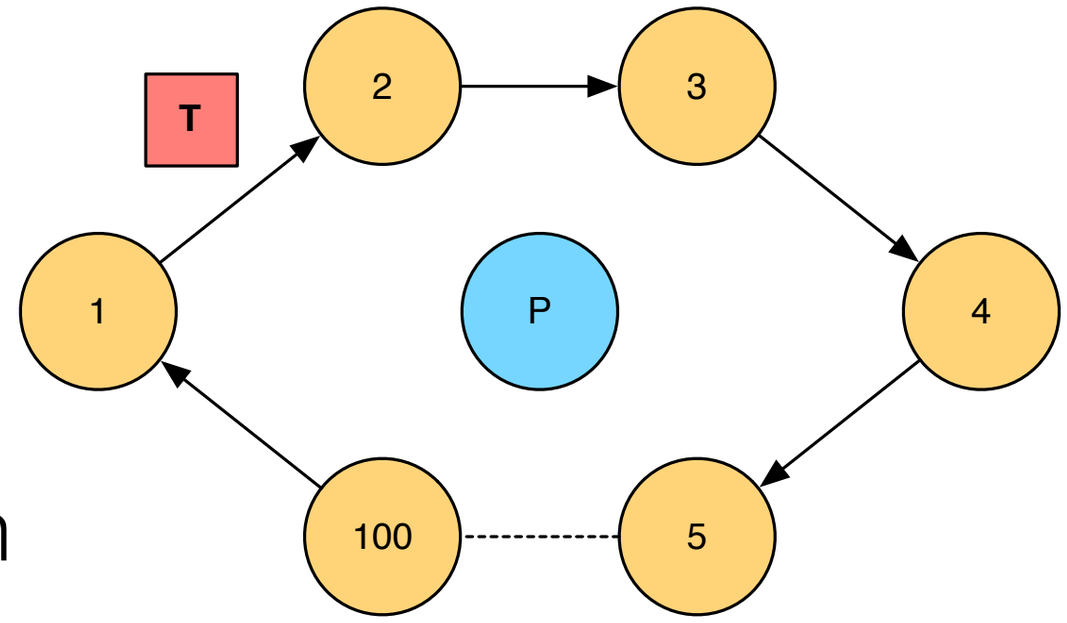
Results

Setup: 12 cores, Linux, GCC 4.8, 1000 x 1000 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
```

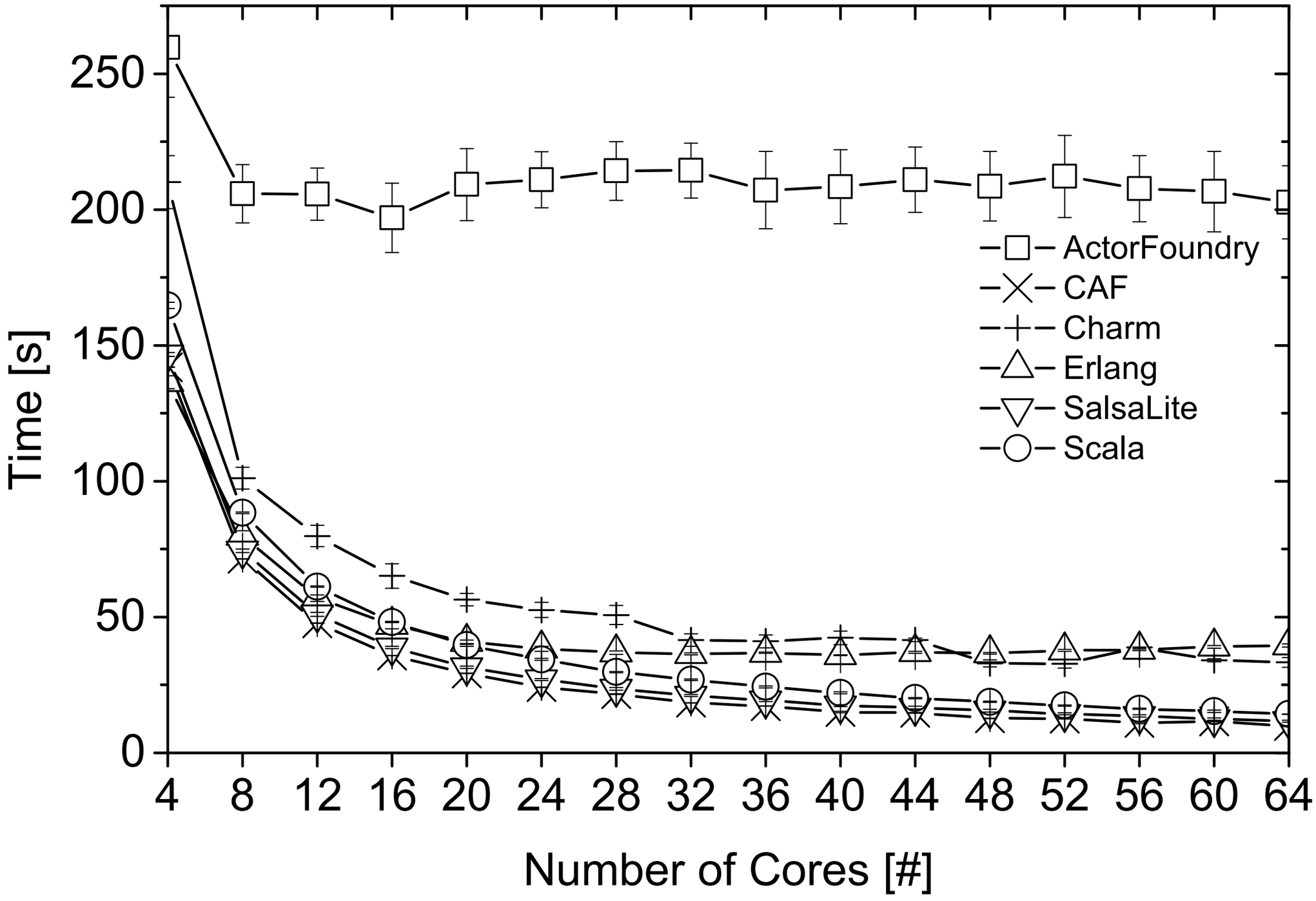
Benchmark #1

Setup #1

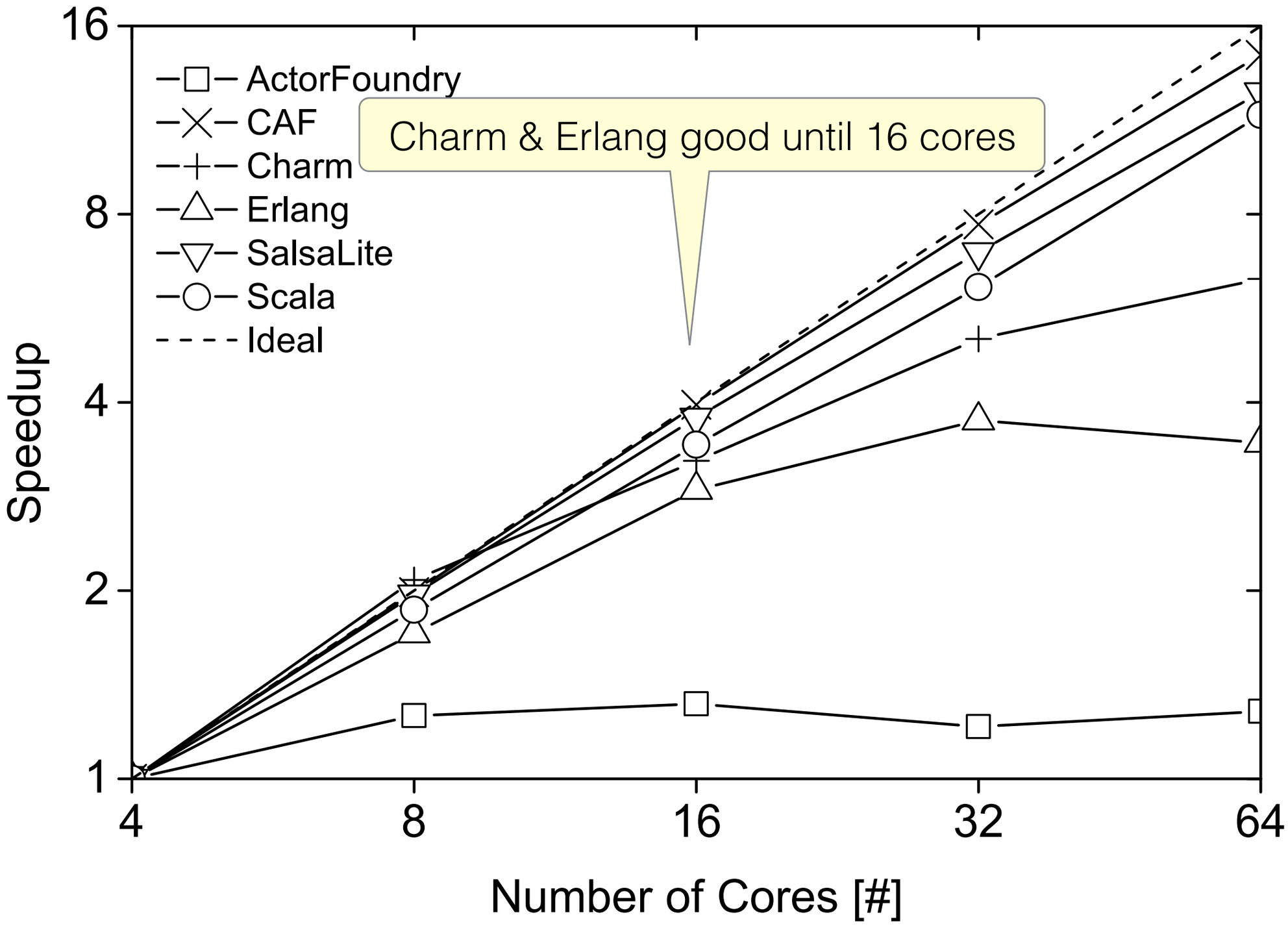


- 100 rings of 100 actors each
- Actors forward single token 1K times, then terminate
- 4 re-creations per ring
- One actor per ring performs *prime factorization*
- Resulting workload: high message & CPU pressure
- Ideal: 2 x cores ⟹ 0.5 x runtime

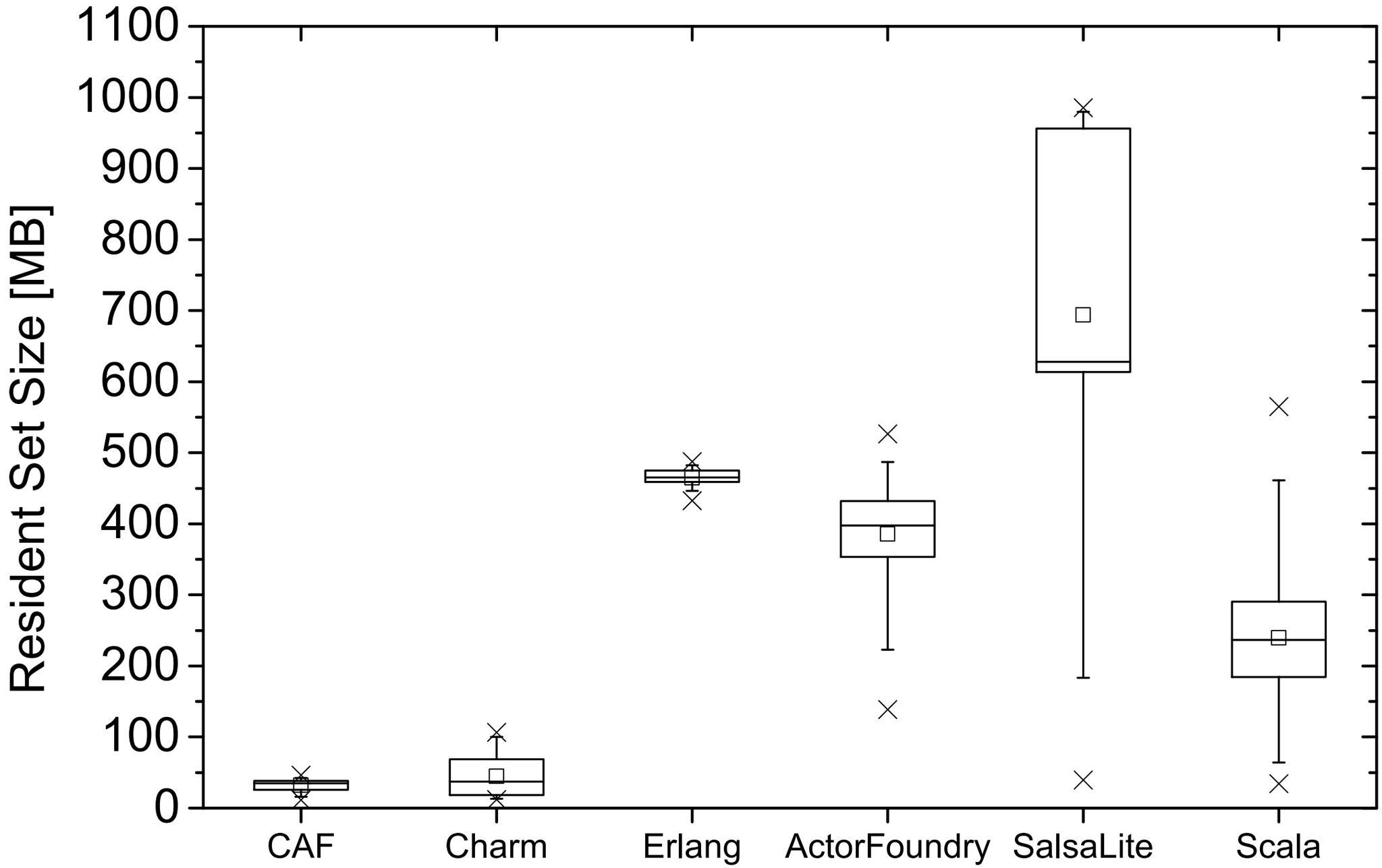
Performance



(normalized)

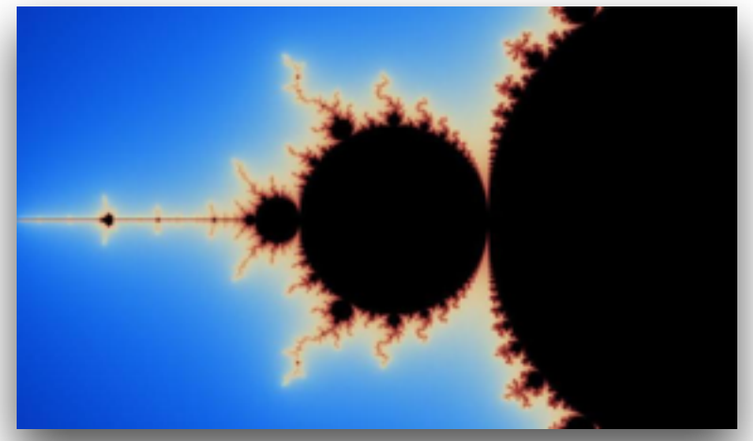

16
8
4
2
1
Speedup
ActorFoundry
CAF
Charm
Erlang
SalsaLite
Scala
Ideal
Charm & Erlang good until 16 cores
4
8
16
32
64
Number of Cores [#]

Memory Overhead



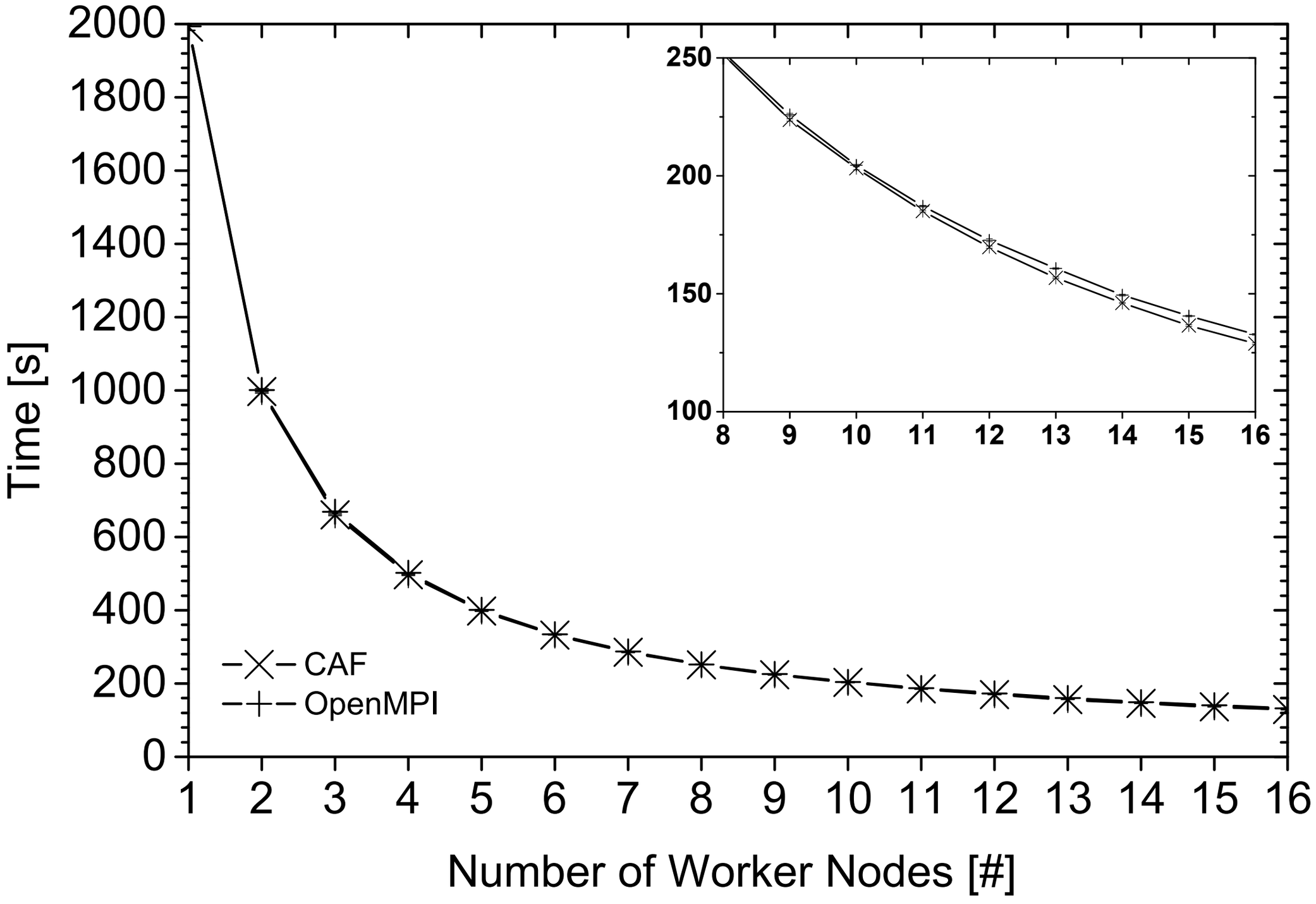
Benchmark #2

Setup #2



- Compute images of Mandelbrot set
- Divide & conquer algorithm
- Compare against OpenMPI (via Boost.MPI)
  - Only message passing layers differ
- 16-node cluster: quad-core Intel i7 3.4 GHz

CAF vs. OpenMPI

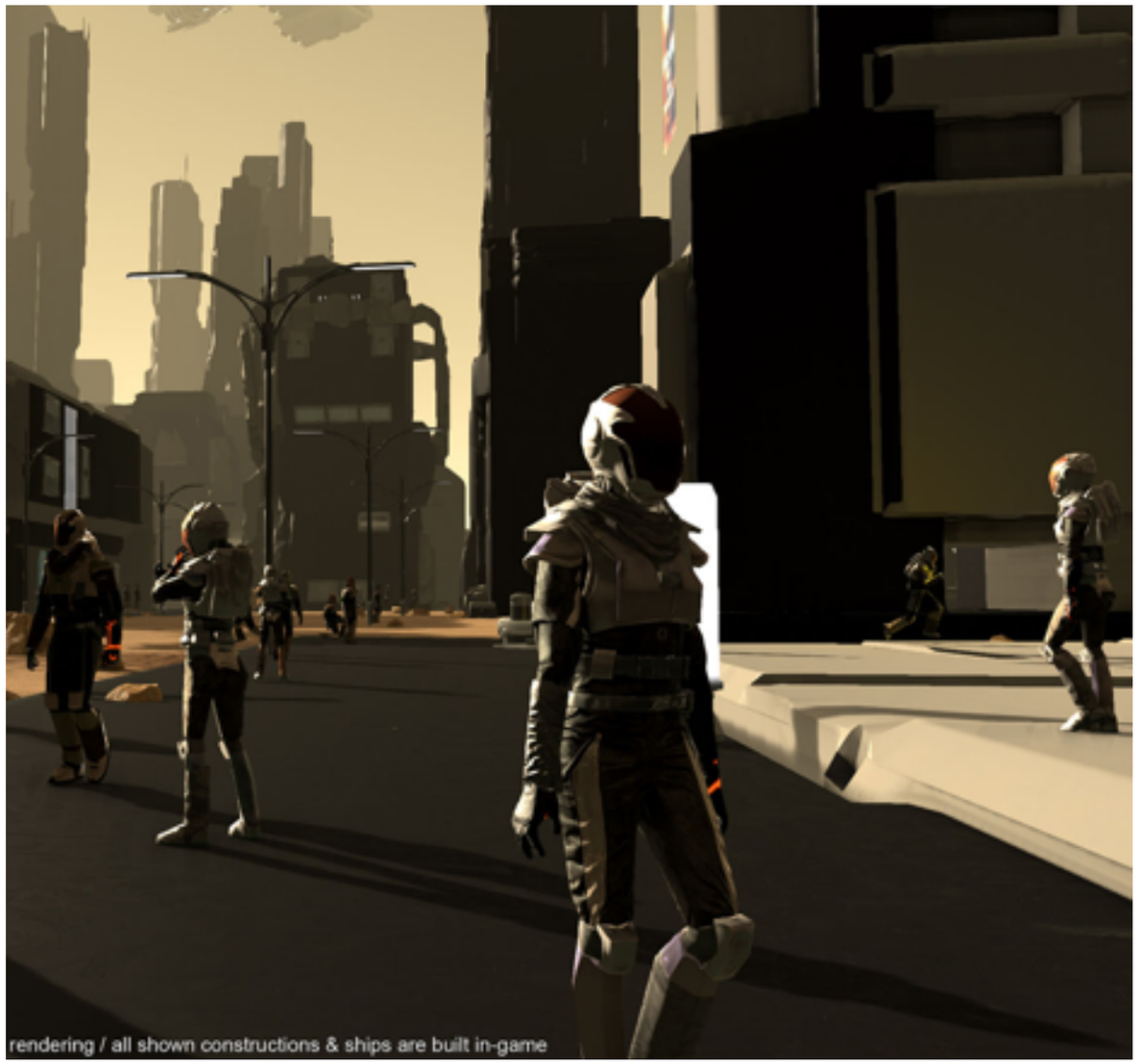


Project

- Lead: **Dominik Charousset** (HAW Hamburg)
  - Started CAF as Master's thesis
  - Active development as part of his Ph.D.
- Dual-licensed: 3-clause **BSD** & **Boost**
- Fast **growing community** (~1K stars on github, active ML)
- Presented CAF twice at C++Now
  - Feedback resulted in type-safe actors
- **Production-grade** code: extensive unit tests, comprehensive CI

CAF in MMOs

- **Dual Universe**
  - Single-shard sandbox MMO
  - Backend based on CAF
  - Pre-alpha
  - Developed at Novaquark (Paris)



http://www.dualthegame.com

CAF in Network Monitors

- Broker: Bro's messaging library
  - Hierarchical publish/subscribe communication
  - Distributed data stores
  - Used in Bro cluster deployments at +10 Gbps

https://github.com/bro/broker

CAF in Network Forensics

- **VAST**: Visibility Across Space and Time
  - Interactive, iterative data exploration
  - Actorized concurrent indexing & search
  - Scales from single-machine to cluster

http://vast.io

Summary

- **Actor model** is a natural fit for today's systems
- **CAF** offers an efficient C++ runtime
- **High-level message passing** abstraction
- **Type-safe APIs** *at compile time*
- **Network-transparent** communication
- **Well-defined failure** semantics

Questions?

http://actor-framework.org

https://github.com/actor-framework

Backup Slides

Sending Messages

- Asynchronous fire-and-forget

```
self->send(other, x, xs...);
```

- Request-response with one-shot continuation

```
self->request(other, timeout, x, xs...).then(
  [=](T response) {
  }
);
```

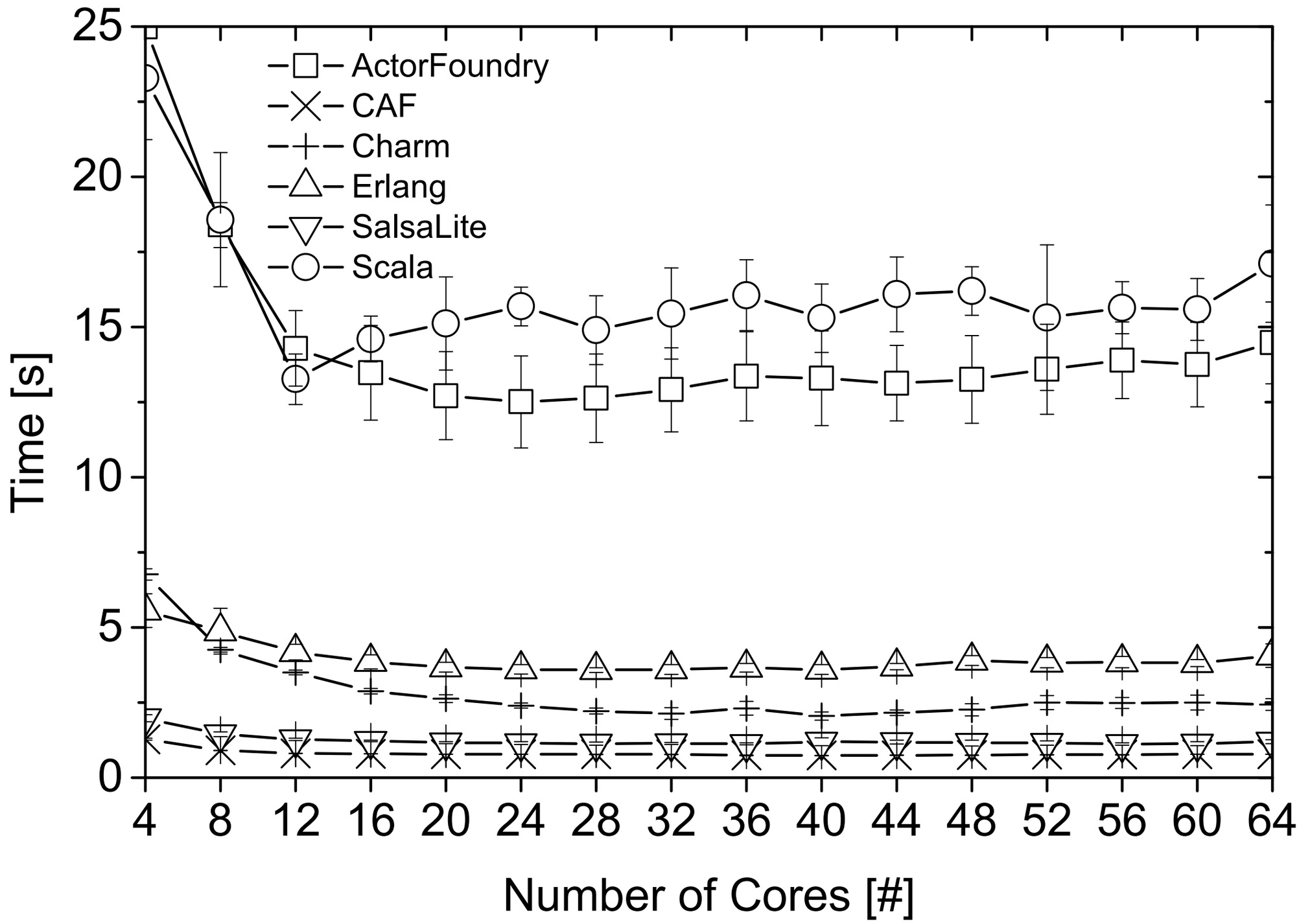
- Transparent forwarding of message authority

```
self->delegate(other, x, xs...);
```

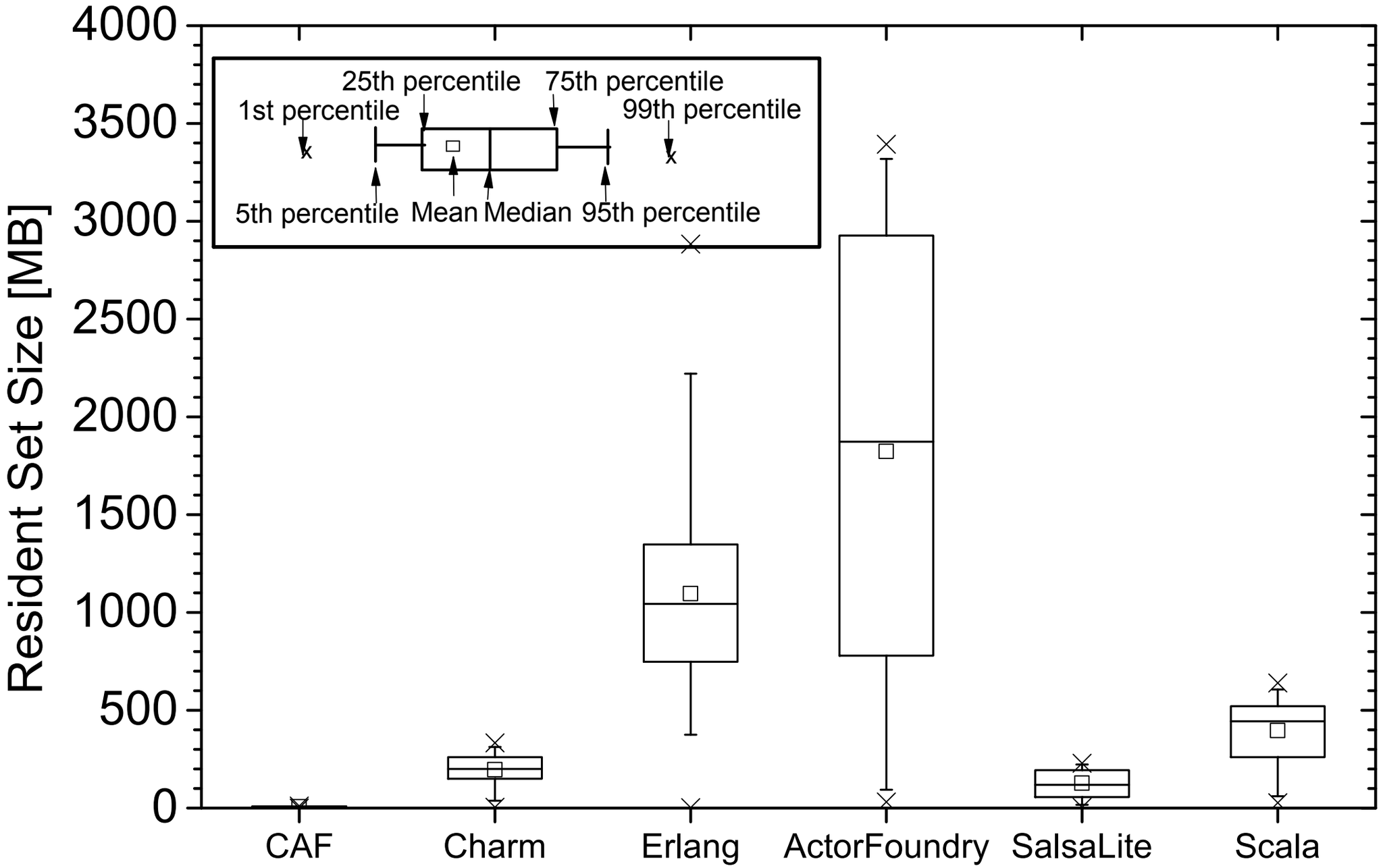
Actors as Function Objects

```
actor a = sys.spawn(adder);
auto f = make_function_view(a);
cout << "f(1, 2) = "
     << to_string(f(1, 2))
     << "\n";
```

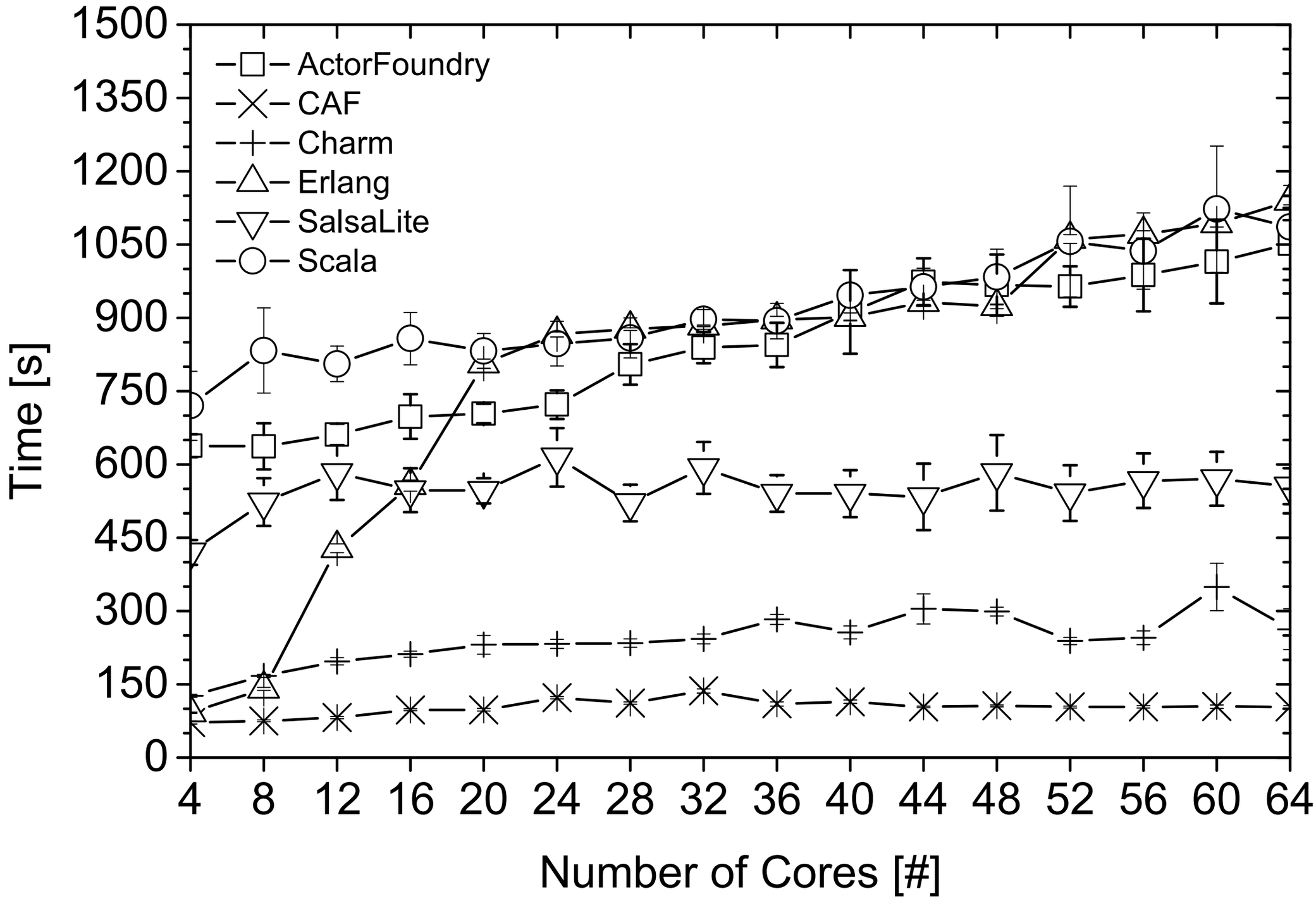
Actor Creation



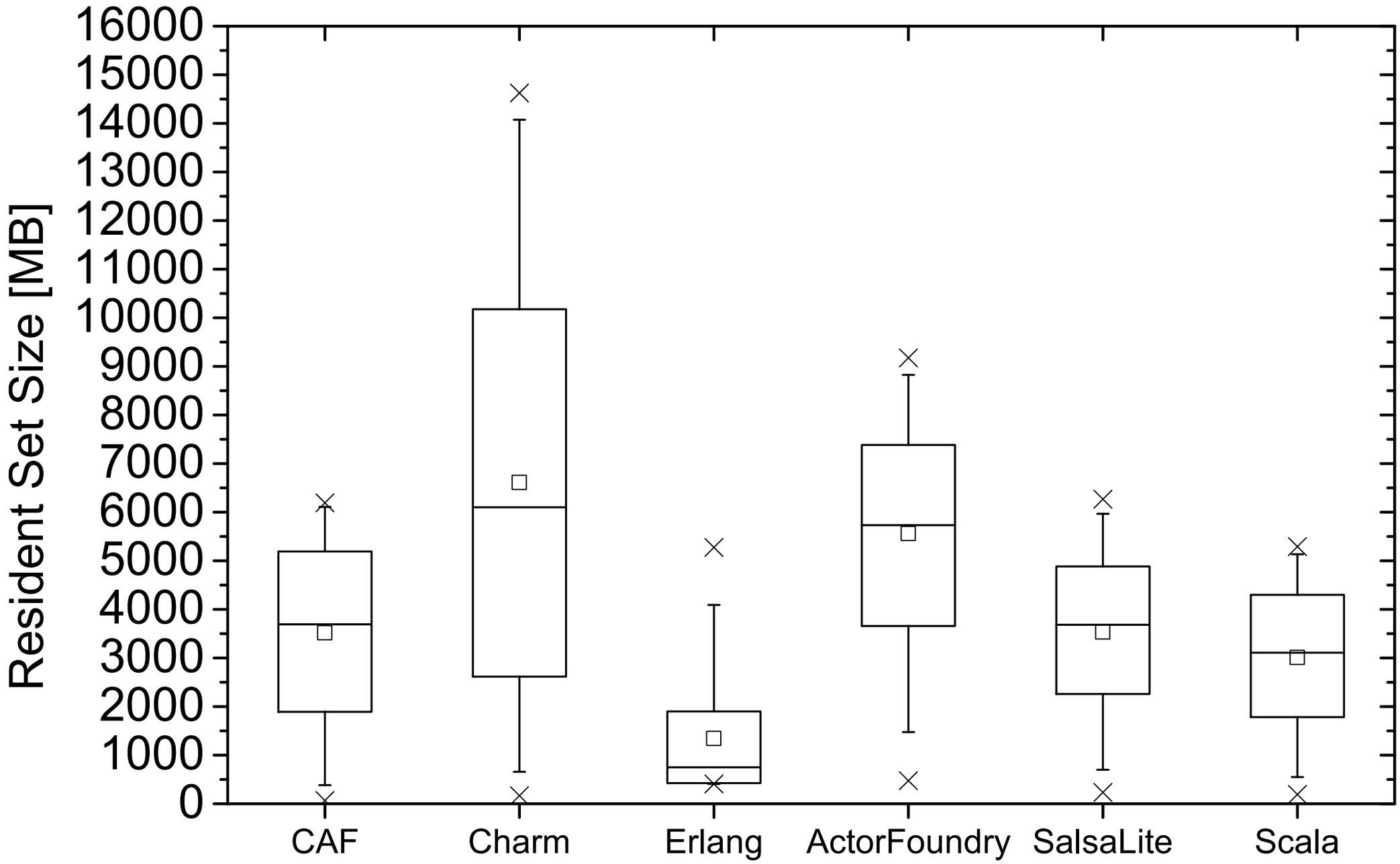
Actor Creation

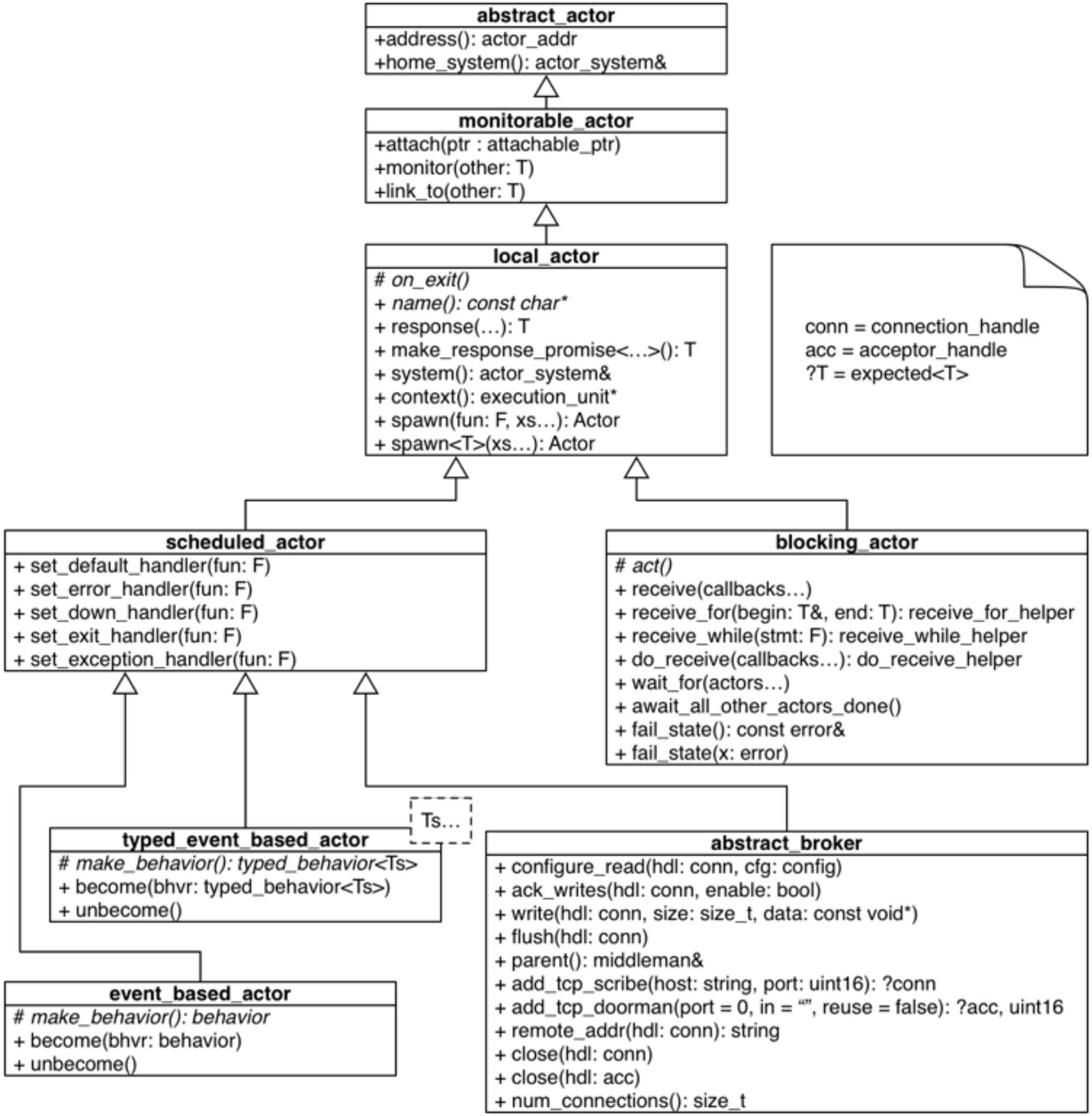


Mailbox - CPU


ActorFoundry
CAF
Charm
Erlang
SalsaLite
Scala
Time [s]
1500
1350
1200
1050
900
750
600
450
300
150
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
Number of Cores [#]

Mailbox - Memory




abstract_actor
+address(): actor_addr
+home_system(): actor_system&
monitorable_actor
+attach(ptr : attachable_ptr)
+monitor(other: T)
+link_to(other: T)
local_actor
on_exit()
+ name(): const char*
+ response(…): T
+ make_response_promise<…>(): T
+ system(): actor_system&
+ context(): execution_unit*
+ spawn(fun: F, xs…): Actor
+ spawn<T>(xs…): Actor
conn = connection_handle
acc = acceptor_handle
?T = expected<T>
scheduled_actor
+ set_default_handler(fun: F)
+ set_error_handler(fun: F)
+ set_down_handler(fun: F)
+ set_exit_handler(fun: F)
+ set_exception_handler(fun: F)
blocking_actor
act()
+ receive(callbacks…)
+ receive_for(begin: T&, end: T): receive_for_helper
+ receive_while(stmt: F): receive_while_helper
+ do_receive(callbacks…): do_receive_helper
+ wait_for(actors…)
+ await_all_other_actors_done()
+ fail_state(): const error&
+ fail_state(x: error)
Ts…
typed_event_based_actor
make_behavior(): typed_behavior<Ts>
+ become(bhvr: typed_behavior<Ts>)
+ unbecome()
abstract_broker
+ configure_read(hdl: conn, cfg: config)
+ ack_writes(hdl: conn, enable: bool)
+ write(hdl: conn, size: size_t, data: const void*)
+ flush(hdl: conn)
+ parent(): middleman&
+ add_tcp_scribe(host: string, port: uint16): ?conn
+ add_tcp_doorman(port = 0, in = "", reuse = false): ?acc, uint16
+ remote_addr(hdl: conn): string
+ close(hdl: conn)
+ close(hdl: acc)
+ num_connections(): size_t
event_based_actor
make_behavior(): behavior
+ become(bhvr: behavior)
+ unbecome()

Implementations	Native Execution	Garbage Collection	Pattern Matching	Copy-On-Write Messaging	Failure Propagation	Dynamic Behaviors	Compile-Time Type Checking	Run-Time Type Checking	Exchangeable Scheduler	Network Actors	GPU Actors	Backend
	Features											
Erlang [13]	✗	✓	✓	✗	✓	✓	✗	✓	✓	✓	✗	BEAM
Elixir [70]	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	BEAM
Akka/Scala [7]	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	JVM
SALSA Lite [62]	✗	✓	✗	✗	✗	✗	✓	✓	✗	●†	✗	JVM
Actor Foundry [2]	✗	✓	✓	✗	✗	✓	✗	✓	✗	✓	✗	JVM
Pulsar [151]	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	JVM
Pony [47]	✓	✓	✓	✗	✗	✗	✓	✓	✗	✓	✗	LLVM
Charm++ [109]	✓	●*	✗	✗	✗	✗	✓	✓	✗	✓	✗	C++
Theron [182]	✓	●*	✗	✗	✗	✓	✗	✓	✗	✓	✗	C++
libprocess [124]	✓	●*	✗	✗	✓	✗	✓	✗	✗	✓	✗	C++
CAF [44]	✓	●*	✓	✓	✓	✓	✓	✓	✓	✓	✓	C++

* Via reference counting, as opposed to tracing garbage collection.
† Only in SALSA, not SALSA Lite.