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

Dominik Charousset Thomas C. Schmidt
Raphael Hiesgen
HAW Hamburg, Dept. Computer Science
dcharousset@acm.org, t.schmidt@ieee.org,
raphael.hiesgen@haw-hamburg.de

Matthias Wählisch
Freie Universität Berlin, Institute of Computer Science
waehlisch@ieee.org

Abstract

Writing concurrent software is challenging, especially with low-level synchronization primitives such as threads or locks in shared memory environments. The actor model replaces implicit communication by an explicit message passing in a 'shared-nothing' paradigm. It applies to concurrency as well as distribution, but has not yet entered the native programming domain. This paper contributes the design of a native actor extension for C++, and the report on a software platform that implements our design for (a) concurrent, (b) distributed, and (c) heterogeneous hardware environments. GPGPU and embedded hardware components are integrated in a transparent way. Our software platform supports the development of scalable and efficient parallel software. It includes a lock-free mailbox algorithm with pattern matching facility for message processing. Thorough performance evaluations reveal an extraordinary small memory footprint in realistic application scenarios, while runtime performance not only outperforms existing mature actor implementations, but exceeds the scaling behavior of low-level message passing libraries such as OpenMPI.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.4 [*Processors*]: Run-time environments

Keywords Actor Model, Concurrent Programming, C++, Message-oriented Middleware, GPGPU Programming

1. Introduction

The majority of programs today is executed in environments of multiple processing units. A key challenge of program development is to appropriately aggregate resources for the sake of code performance, execution efficiency, and particular application needs. Multi-core CPUs have become an integral part of commodity hardware even in mobiles. Heterogeneous hardware components like graphics processing units (GPUs) and embedded controllers contribute powerful capacities to end systems, while novel computing paradigms arise in emerging distributed eco systems like cloud computing [5, 10] and mobile crowd sourcing [21]. All these scenarios rely on concurrency [12], many also require distribution. Still the dominant part of current applications is written in some popular imperative language [28].

Imperative programming languages such as C, C++, or Java do not provide concurrency semantics. They were developed before the 'multi-core revolution' started and thus originally aimed at single-core processor machines. Threading libraries were added on top of existing languages that allow to start multiple *threads of execution* within a process. However, dealing with concurrency is challenging, especially in shared memory environments where parallel access to process-wide memory easily leads to *race conditions*. The performance and scalability of hand-written synchronization for avoiding race conditions depends on the implementation strategy. Coarse-grained locking is simple, but easily causes queuing and scalability issues, whereas fine-grained locking increases scalability but also complexity and error-proneness due to lock order, for example. Additionally, time-dependent errors make it virtually impossible to verify a concurrent application by systematic testing [13].

A powerful approach to the problems of concurrency and distribution has been formulated in the actor model by Hewitt, Bishop, and Steiger [16]. This formalism describes concurrent entities – 'actors' – that execute independently, do not share state, and communicate by asynchronous message passing. Because actors are self-contained and do not rely on shared resources, race conditions are avoided by design. The message passing communication style facilitates a transparent deployment and applies to (1) concurrency, if actors run on the same machine, (2) heterogeneous environments, if actors on the same machine are bound to different memory regions and processing units, and (3) distribution, whenever actors run on different hosts connected via the network. Actor-based languages like Erlang [3] and frameworks such as Akka [30] or Kilim [25] have been bound to specific niches or use vendor specific APIs (e.g., Casablanca [20]). One major objective of the present work is to make actor programming accessible to a wider community and to broaden its range of applications. We therefore extended the actor model to wider applicability and designed and implemented an actor framework for C++ from scratch.

This paper makes three main contributions. First, we present a design for a C++ software platform based on the actor model. Our design includes a lightweight, lock-free, and network-transparent message passing system that is implemented without context switching. Second, we present an approach to transparently integrate heterogeneous hardware components using OpenCL by deducing the messaging interface from the kernel signature. Third, we demonstrate experimentally that our approach is practical and efficient enough to outperform mature implementations of the ac-

AGERE '13, Month d–d, 20yy, City, ST, Country.
Copyright © 2013 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/

tor model, as well as low-level message passing libraries such as OpenMPI in several scenarios and key aspects. We highlight key achievements of our platform in Section 2.

The remainder of this paper discusses relevant design aspects along with related work in Section 3. Section 4 details the design decision and major implementation choices. The practical performance evaluation is presented in Section 5. Finally, Section 6 discusses the lessons learned and Section 7 concludes.

2. The Case for Actors

This part highlights arguments for the actor platform from a programming and a performance perspective. Consider the following class.

```
class KeyValueStore {
 public:
  void set(Key k, Value v) {
   // ...
  }
  Value get(Key k) const {
   return ...;
  }
 private: // ... implementation details
};
```

When accessible by multiple threads in parallel, the class has to be implemented in a thread-safe manner. A simple approach is to guard both member functions using a mutex. This approach does not scale well, mainly because readers block other readers. More scalable approaches require a specific synchronization protocol that is based on recursive or shared mutexes, for example. The following source code illustrates the definition of an actor in our software platform.

```
become (
 on(atom("set"), arg_match)
 >> [=](Key k, Value v) { /* ... */ }
 on(atom("get"), arg_match)
 >> [=](Key k) {
  reply(...);
 }
);
```

Actors can be programed without knowledge about concurrency primitives. At the same time, our actor implementation supports massively parallel access (cf. § 5.3). In the example, key requests are sequentially processed without further coordination, as there is no intra-actor concurrency. For a further increase of parallelism, actors can explicitly redistribute tasks to a set of "workers". Our message passing interface uses so-called atoms to identify specific operations instead of member function names. Atoms do not cause string comparison at runtime, as they are converted to integers by using a hash function at compile time. The following code shows the caller side of our example.

```
sync_send(server, atom("get"), key).then(
 [=](Value val) {
  cout << key << " => " << val << endl;
 }
);
```

Our software platform provides network-transparent messaging. To highlight its performance in a concurrent and a distributed system, we have implemented an algorithm to calculate a fixed number of the Mandelbrot set. The computation was distributed by using our library (`libcppa`) for actor programming in C++, and OpenMPI, a low-level message passing library optimized for high performance. Since both programs share one C++ implementation for the calculation, the measurements reveal the overhead added by the distribution technologies in use. Hence, this setup discloses the trade-offs in performance developers make when opting for a high-level abstraction like the actor model instead of for low-layer primitives.

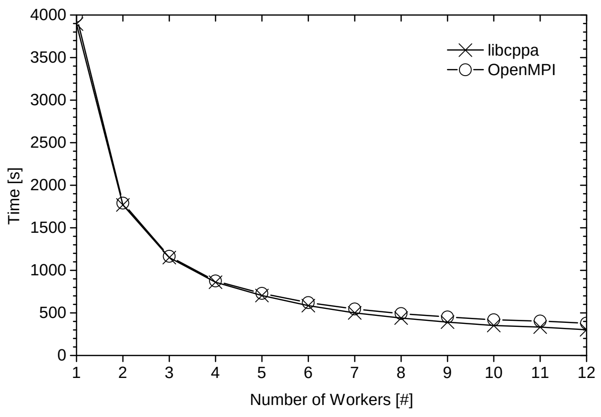


Figure 1. Sending and processing time for 778 images of the Mandelbrot set in a concurrent system consisting of 12 nodes using `libcppa` and OpenMPI

Figure 1 shows the runtime results for concurrent setups as functions of available worker nodes. In this evaluation, we have used one host machine running 12 virtual machines as worker nodes. The additional transmission and processing overhead of `libcppa` is only visible for up to three workers. The multiplexing capabilities of `libcppa` surpass OpenMPI for four and more workers, which indicates a better scalability of `libcppa`.

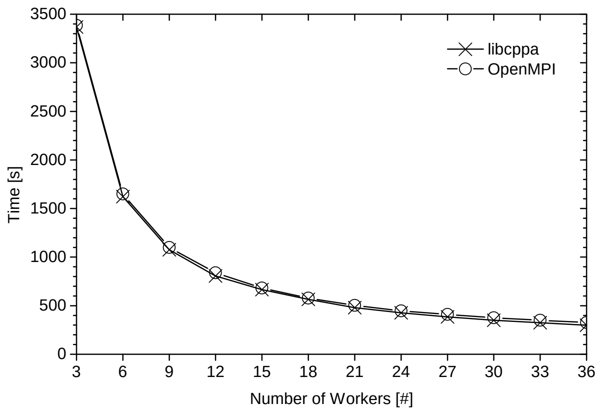


Figure 2. Sending and processing time for 2314 images of the Mandelbrot set in a distributed system consisting of 36 nodes using `libcppa` and OpenMPI

The results for our distributed scenario are shown in Figure 2. In this setup, we have used three host machines running 12 virtual machines each, while the problem size was increased by a factor of ≈ 3. To achieve an evenly distributed work load, we added worker nodes in increments of three, i.e., one worker at each host machine. As in our previous results, the runtime advantage of `libcppa` over OpenMPI increases with the number of workers: from ≈ 3 s on three nodes up to ≈ 23 s on 36 nodes.

These results clearly illustrate that actors created from `libcppa` do not impose a performance penalty. Hence developers do not need to choose between a high level of abstraction on the one hand, and runtime performance on the other. An efficient implementation of the actor model can rather outperform low-level approaches.

3. Background and Related Work

Concurrent software components often need to share or exchange data. In shared memory environments, this communication is implicit via shared state. However, unsynchronized parallel access to shared memory segments easily leads to erroneous behavior caused by race conditions. To prevent parallel or interleaved execution of so called *critical sections*, developers have to implement a synchronization protocols by using low-level primitives such as locks and condition variables. This is inherently error-prone, and correctly implementing critical sections requires expert knowledge about compiler optimizations, out-of-order execution, and other. [19]. Furthermore, a central software component that is exposed to frequent parallel access should best not rely on locking, since a critical section is a likely performance bottleneck. Lock- and wait-free algorithms [14] can scale in a multiprocessor environment, but are significantly more complex [9].

Designing suitable locking strategies or lock-free algorithms are not the only challenges developers face on parallel hardware. Applications with good performance on a uniprocessor machine may experience a performance degradation on multiprocessor platforms, e.g., due to *false sharing*. False sharing occurs whenever two or more processors are repeatedly writing into a memory region that is mapped to more than one processor cache. This mutually invalidates the caches, even if the processors are accessing distinct data, and severely slows down program execution [29].

3.1 Concurrent Programming and Actors

Higher-level abstractions following a message passing or transactional memory paradigm have been developed that do not require synchronization of readers and writers. They established programming models that are free of race conditions. Transactional memory can be implemented in hardware [15] or software [24]. However, transactional memory applies neither to event-driven workflows, nor to communication between heterogeneous hardware components, nor to distributed systems. Message passing, on the other hand, has proven to scale in multiprocessor environments as well as in distributed scenarios such as cluster computing. In the high-performance domain, message passing systems based on MPI have been in use for decades [11].

The actor model [16] is based on the message passing paradigm, but raises the level of abstraction even further. It not only describes how software components communicate, but also characterizes the communicating components, the actors themselves. Based on this model, concurrent and distributed systems can be composed of independent modules [1] that are open for communication with external components [2]. In addition, the actor model addresses reliability and fault tolerance in a network-transparent way [4]. Hence, developers no longer need to optimize their code depending on the target platform. Deployment on parallel, heterogeneous, or distributed resources can be delegated to the runtime environment, since the deployment does not affect the logical structure of an actor program.

3.2 Message Processing

Agha [1] introduced mailbox-based message processing in its seminal modeling work on actors. A mailbox is a FIFO ordered message buffer that is only readable by the actor owning it, while all other actors are allowed to enqueue new messages. Mailboxes exclusively enable communication between actors, as no state is shared. Implementation concepts of mailbox management divide into two categories.

In the first category, an actor iterates over messages in its mailbox. On each receive call, it begins with the first but is free to skip messages. Messages can be automatically postponed, if the behavior of an actor is defined as a partial function [12]. As actors can change their behavior in response to a message, a newly defined behavior may apply to previously skipped messages. A message remains in the mailbox until it is eventually processed and removed as part of its consumption. Erlang [4] is the classical example for this category of message processing.

The second category of actor systems follows a more restrictive message processing scheme. A message handler is invoked exactly once per message with the specific behavior of the actor. An untreated message cannot be recaptured at a later time, even though some systems allow to change the message handler at runtime. Consequently, actors are forced to handle messages in the order of arrival. The examples of SALSA [31], Akka [30], Kilim [25], and Retlang [23] fall in this category.

In our work, we followed the first approach. The major reasons for this are the abilities to prioritize messages and to wait for a response prior to returning to the default behavior. In this context, pattern matching has proven useful and very effective to ease definition of partial functions used as message handlers [3]. Thus, we provide pattern matching for message handling as a domain-specific language in our system.

3.3 Fault Propagation

Fault-tolerant distributed systems require strong mechanisms of fault monitoring and control. The actor model foresees actors to monitor each other [16]. Whenever an actor fails, an exit message is sent to all actors that monitor it. This monitoring can be bidirectional, and actors that established such a strong relationship are called *linked*. Linked actors form a subsystem in which errors are propagated via exit messages.

Based on linking, developers can build subsystems in which all actors are either alive or have collectively failed. Each subsystem can include one or more actors that survey working actors and re-create failing workers. In proceeding this way, hierarchical, fault-tolerant systems such as Erlang's *supervision trees* [4] can be built. Newer implementations of the actor model (e.g., Kilim and Akka) have adopted the Erlang model of error propagation, as it has proven very effective, elegant and reliable [22] in tightly coupled domains. We adopted this well-established fault propagation model as a starting point, but are aiming for and extension of higher flexibility that may well server loosely coupled and non-hierarchical domains.

4. A Software Platform for Actor Programming in C++

In this section, we present our software platform bundled in the C++ library `libcppa`. We discuss its design along with selected implementation decisions in detail. The software has been released as open source under the LGPL 2.1 license.[1]

4.1 Requirements

The following design requirements account for fast program execution as well as developer efficiency by providing the highest level of abstraction possible without sacrificing performance.

Scalability: In the context of multi-core processors, scalability requires to split the application logic into many independent tasks that can be executed in parallel. Developers shall be enabled to create a significant number of short-lived actors without performance penalty. In contrast, the canonical transfer of tasks to threads does not scale well in a short-lived environment, since the effort of creation and destruction of threads often outweighs the benefit of parallelization. Hence, actors must have small memory footprint as well as low scheduling overhead.

[1] See `http://www.libcppa.org`.

Distribution Transparency: The network layer of `libcppa` shall manage all communication requests, thereby hiding complexity of the underlying communication protocols and deployment. Furthermore, the addressing of an actor shall rely on a common interface for local and remote actors so that applications will experience a uniformly transparent access and easily rescale at runtime.

Message Handling and Processing: Messages shall (a) be garbage collected, (b) not be limited to particular types, and (c) provide pattern matching. Requirement (a) is owed to the experience that manual memory management in concurrent systems is error-prone and thus impractical, while the alternative approach of copying a message for each single recipient, leads to suboptimal performance if a message has multiple recipients. Requirement (b) reflects the common experience that message passing with restricted types is of limited use in practice. However, unrestricted messaging requires efficient and expressive facilities such as pattern matching (c), because message handling is a continuously recurring task to implement.

4.2 Key Concepts in Design and Implementation

In this section, we go into selected details of relevant technological and algorithmic choices that shall fulfill the requirements and provide a modern C++ API design.

4.2.1 Copy-On-Write Messaging using Tuples

The message passing implementation of `libcppa` uses tuples with call-by-value semantics. This may lead to multiple copies of a tuple when sending to more than one actor. Copy-on-write is an optimization strategy to minimize copying overhead, so that a tuple can be shared among several actors as long as all participants only demand read access. An actor copies the shared tuple when it requires write access and is only allowed to modify its own copy. Thus, race conditions cannot occur and each tuple is copied only if needed. This also implements garbage collection as unreferenced tuples are deleted automatically. We have used an atomic, intrusive reference counting implementation that adds only a negligible runtime overhead.

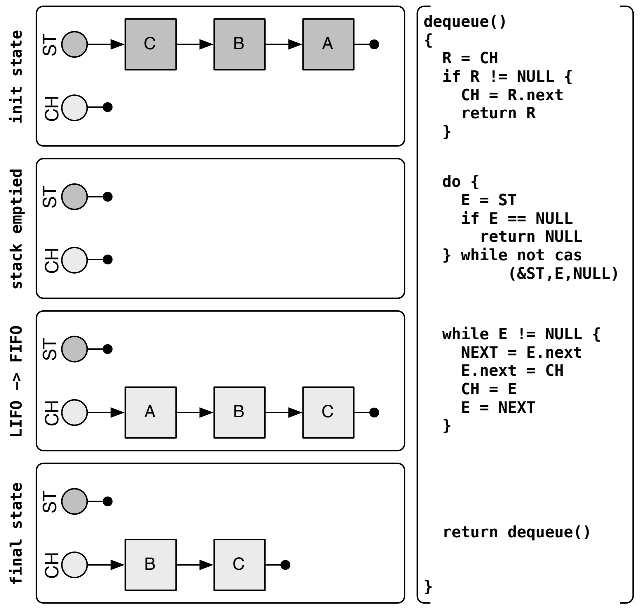
4.2.2 Mailbox Algorithm

The message queue or *mailbox* implementation is a critical component of any message passing system. All messages sent to an actor are delivered to its mailbox, which acts as a shared resource whenever an actor receives messages from multiple senders in parallel. Thus, the overall system performance, foremost its scalability depends significantly on the selected algorithm.

A mailbox is a single-reader-many-writer queue. It is exposed to parallel write access, but only the owning actor is allowed to dequeue a message. Hence, the dequeue operation does not need to support parallel access. We have combined a lock-free stack implementation with a FIFO ordered queue as internal cache. A lock-free stack can be implemented using a single atomic compare-and-swap (CAS) operation. It does not suffer from the so called ABA problem of concurrent access that can corrupt states in CAS-based systems [17] as the enqueue operation only needs to manipulate the *tail* pointer. However, without reordering the dequeue operation would have to traverse the (LIFO-sorted) stack in order to find the oldest element.

Figure 3 shows the dequeue operation of our mailbox implementation. It always dequeues elements from the FIFO ordered cache (CH). The stack (ST) is emptied and its elements are moved in reverse order to the cache whenever it drains. Emptying the stack can be done by a single CAS operation as it only needs to set ST to NULL.

Our mailbox has complexity $O(1)$ for enqueue operations, while the dequeue operation has an average runtime of $O(1)$, but a



```
dequeue()
{
  R = CH
  if R != NULL {
    CH = R.next
    return R
  }

  do {
    E = ST
    if E == NULL
      return NULL
  } while not cas
        (&ST,E,NULL)

  while E != NULL {
    NEXT = E.next
    E.next = CH
    CH = E
    E = NEXT
  }

  return dequeue()
}
```

Figure 3. Dequeue operation in a cached stack (ST = **S**tack **T**ail, CH = **C**ache **H**ead)

worst case of $O(n)$, where n is the maximum number of messages in the stack. Concurrent access to the cached stack is reduced to a minimum and both enqueueing and dequeueing perform only a single CAS operation. Our performance measurements (cf., Section 5) show that this lock-free implementation enables `libcppa` to utilize hardware concurrency in N:1 communication scenarios more efficiently than common implementations of the actor model.

4.2.3 Pattern Matching for Tuples using Partial Functions

C++ does not provide pattern matching facilities, and a general pattern matching solution for arbitrary data structures would require a language extension. Hence, we decided to restrict pattern matching to tuples, which can be achieved by an internal domain-specific language (DSL) approach. A match expression, i.e., the definition of a partial function, begins with a call to the function `on` that returns an intermediate object providing the member function `when` and the operator ">>". The right-hand side of the operator denotes a callback—usually a lambda expression—which should be invoked after a tuple matches the types given to `on`. This is shown in the example below.

```
on<int>() >> [](int i) { /*...*/ }
```

The result of operator ">>" is a partial function that is defined for the types given to `on`. Additionally, guards can be used to constrain a given match statement by using placeholders or values, as the following example illustrates.

```
on(42) >> [] {
 // matches only the integer value 42
},
on<int>().when(_x1 % 2 == 0) >> [] {
  // matches even integer values (except 42)
},
on<int>() >> [] {
 // matches odd integer values; not invoked
 // if a previous rule matches first
}
```

Guard expressions are a lazy evaluation technique. The placeholder `_x1` is substituted with the first element of a given tuple.

A comma separated list of partial functions results in a single partial function that sequentially evaluates its subfunctions. At most one callback is invoked, since the evaluation stops at the first match. Hence, the second lambda expression in the example above can safely assume that it is invoked if and only if the integer is not even. The type "anything" can be used as wildcard expression to match any number of any types. For example, "on<int,anything>()" matches all tuples with an integer as first element. However, repeating types in both the callback signature and the template parameter list is redundant, as these types can be deduced automatically. For this purpose, we have introduced the keyword-like constant arg_match that can be used as last argument to on and causes the compiler to deduce all further types from the signature of the callback, as illustrated in the example below.

```
partial_function fun {
 on(atom("add"), arg_match) >> [](int v) {
  // matches the types <atom_value, int>
  // and is only invoked if the first
  // value is the atom 'add'
 },
 on_arg_match >> [](int a, int b) {
  // matches the types <int, int>
  // on_arg_match is a convenience
  // keyword-like value
  // that is equal to on(arg_match)
 }
};
```

Our DSL-based approach has more syntactic noise than a native support within the programming languages itself, for instance when compared to functional programming languages such as Haskell or Erlang. However, we only use ISO C++ facilities, do not rely on brittle macro definitions, and our approach only adds negligible—if any—runtime overhead by using expression templates [32].

4.2.4 Actor Semantic as Internal Domain-Specific Language for C++

The keyword self is an essential ingredient of our design. From a user's point of view, the keyword identifies the running actor like the implicit this pointer identifies an object within a member function. Unlike this, though, self is not limited to a particular scope. Furthermore, it is not just a pointer, but needs to execute implicit conversion on demand when used from a context that was not explicitly created as an actor. In that case, a new object containing a mailbox, link-list, etc. is created. It converts the given thread to a thread-mapped actor. This approach ensures a consistent programming model in which "everything is an actor". The self pointer is used implicitly, whenever an actor calls functions like send, but can be accessed to use more advanced actor operations such as linking to another actor, i.e., by calling self->link_to(other).

4.2.5 Cooperative Scheduling of Actors

An actor library needs to schedule actors in an efficient way. An ideal way to ensure *fairness* in an actor system requires preemptive scheduling. A fair system would guarantee that no actor could starve other actors by occupying system resources. However, preemptive scheduling requires hardware interrupts to switch between running tasks and thus can only be implemented, if unrestricted access to hardware or kernel space is granted. For obvious reasons, a full-fledged operating systems cannot allow unrestricted hardware or kernel space access to a userspace application like our library.

In general, userspace schedulers can only implement cooperative scheduling. We decided to implement an opt-out cooperative scheduling, so that developers can choose to execute an actor in its own thread in case it relies on blocking system calls that may starve other actors in a cooperative scheduling. All cooperatively scheduled actors run in a thread pool that is pre-dimensioned according to the number of cores discovered at runtime. The elements in the shared job queue of the thread pool, i.e., the actors in ready state, are aligned according to the cache line size of the hardware platform to avoid false sharing. Our benchmarks, in particular the emulation of a realistic use case scenario, illustrate that our cooperative scheduling implementation performs comparably with industrial strength implementations such as Akka.

4.2.6 Transparent Integration of Heterogeneous Hardware Components

With the advent of GPGPU programming, it became a crucial factor for a broad range of applications to make use of the heterogeneous computing platforms found in modern hardware deployments. This demand has lead to the development of the open standard OpenCL [26]. In OpenCL, developers provide an implementation of an algorithm, the so-called *kernel*, in a C dialect that is compiled for the detected hardware at runtime. The following code example shows the prototype of an OpenCL kernel to multiply two matrices.

```
__kernel void matrix_multiply(
                 __global float* matrix1,
                 __global float* matrix2,
                 __global float* output);
```

By convention, the last parameter is the output parameter. When instantiating this kernel at runtime to create an OpenCL *program*, all three dimensions, i.e., the number of elements in matrix1, matrix2, and output, must be defined. In order to execute matrix_multiply, one needs to encapsulate the function call along with the parameters as a task and then enqueue this task to an OpenCL *command queue*. OpenCL offers a callback-based API as well as a blocking API to await the completion of a task.

The task-based workflow of OpenCL is a natural fit to the actor model. Naturally, an OpenCL program can be regarded as an actor. It awaits input parameters and then produces results. In this exact way, libcppa creates a message passing interface for OpenCL programs, as shown in the following example.

```
spawn_cl<float*(float*,float*)>(
  source, "matrix_multiply", {size, size});
```

The function spawn_cl expects the signature of the OpenCL kernel as a template parameter, normalized to a form with a result type instead of an implicit output parameter. The argument source is a string containing the source code of the kernel. The second argument is the name of the kernel. Finally, spawn_cl expects the dimensions of the input parameters. The invocation example shown above creates an actor that receives two arrays, each consisting of size · size (dimension on the x-axis multiplied with the dimension on the y-axis) elements, and replies a new array containing the resulting matrix. The matrices are represented in one dimension, since OpenCL does not support multi-dimensional arrays. The function spawn_cl also provides several overloads for fine-tuning the OpenCL behavior, or to perform data transformation. The latter allows to hide the kernel signature by providing a different interface to other actors. This is particularly useful to integrate OpenCL actors into an existing application.

4.2.7 Message Processing

An actor uses become to set its behavior in response to incoming messages. The selected behavior is then executed until it is replaced by another call to become or the actor finishes execution. Actors can be implemented using functions or classes. Class-based actors either subtype event_based_actor and implement the pure virtual member function init, or subtype sb_actor ("State-Based Actor") and provide an init_state member variable of type behavior, as shown below.

```
struct printer : sb_actor<printer> {
  behavior init_state = (others() >> [] {
    cout << to_string(self->last_received())
         << endl;
  });
};
```

The class `sb_actor` uses the Curiously Recurring Template Pattern [8], where the derived class must be provided as template parameter. This technique allows `sb_actor` to override the `init` member function as `become(this->init_state)`.

An actor can set a new behavior by calling `become` with the `keep_behavior` policy to wait for the required message and then return to the previous behavior by using `unbecome`, as shown in the example below. An actor finishes execution for normal exit reasons whenever the behavior stack is empty after calling `unbecome`. It is worth mentioning that the original actor model did not include an `unbecome` primitive. Nevertheless, it was added by recent implementation such as Akka to provide a convenient way of defining nested receive operations as shown in the example below. The default policy of `become` is `discard_behavior` that causes an actor to override its current behavior. The optional policy flag must be the first argument of `become`.

```
// receives {int, float} sequences
void testee() {
  become (
    on<int>() >> [=](int value1) {
      become (
        // the keep_behavior policy stores
        // the current behavior on the
        // behavior stack to be able to
        // return to this behavior later
        // on by calling unbecome()
        keep_behavior,
        on<float>() >> [=](float value2) {
          cout << value1 << " => "
               << value2 << endl;
          unbecome();
        }
      );
    }
  );
}
```

We provide two ways of passing messages between actors. The function `send` models asynchronous communication, while `sync_send` models synchronous communication by using unique request identifiers. The response message to a previous request can be received by using a “one-shot handler” that unambiguously matches the response to the request by using `sync_send`. We provide a continuation-like API as illustrated by the following example.

```
sync_send(testee, atom("get")).then(
  on_arg_match >> [=](const string& str) {
    // handle str
  },
  after(chrono::seconds(30)) >> [=] {
    // handle timeout
  }
);
```

5. Performance Evaluation

In this section, we compare the runtime behavior of our software with common implementations of the actor model. Our performance analysis includes local and distributed actors. The source code of all benchmark programs are published online at `https://github.com/Neverlord/cppa-benchmarks`.

5.1 Basic Measurement Setup and Metrics

Erlang and Scala are currently the most relevant languages for actor programming. We use their implementation as reference for concurrent computation. For Scala, we consider the Akka library [30], that is part of Scala’s standard distribution. In detail, our benchmarks are based on the following implementations of the actor model: (1) C++ with `libcppa` (*cppa*), Scala 2.10.0 with the Akka library (*scala*), and (3) Erlang in version 5.9.1 (*erlang*). `libcppa` has been compiled with optimization level O4 of GNU C++ compiler version 4.7.2. Scala runs on a JVM configured with a maximum of 4 GB of RAM.

To quantify the characteristic performance of the actor implementations, we measure (a) the actor creation overhead, (b) the mailbox performance in N:1 communication scenarios, (c) the runtime under significant workload, and (d) the performance in a high-performance distributed computing application. We concentrate on the runtime performance and memory footprint. The benchmarks have been conducted on a Linux host that consists of two hexa-core Intel Xeon processors with 2.27 GHz. To reflect local concurrency, we vary the number of used CPU cores from 2 to 12. For the distributed scenario, we deploy a LAN scenario with nine virtual machines running Linux. This reduced complexity in network topology allows to explore the basic properties of our design and implementation. Each scenario is sampled with the same parameter settings until it is converged. We average the results over all samples with the same settings. For the timing behaviour, the error bars in the subsequent graphs show the 95% confidence interval to represent the variability of the measurements. The memory consumption is visualized by box plots due to the (partly) fluctuation nature of this measurement.

5.2 Overhead of Actor Creation

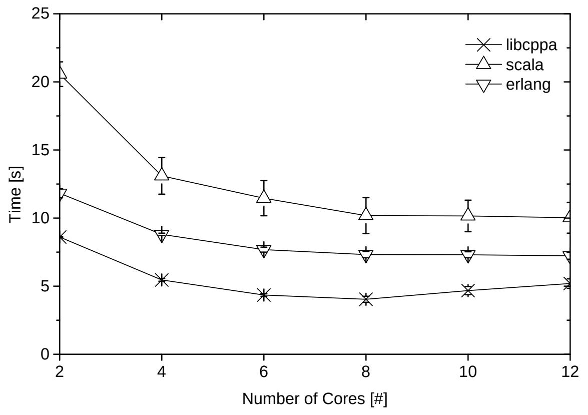
Our first benchmark measures the overhead of actor creation. The following pseudo code illustrates the recursive creation of a predefined number of actors.

```
main(X):
 spawn(testee, self) ! {spread, X}
 receive: {result, Y} => assert(2^X == Y)
testee(Parent):
 receive:
  {spread, 0} => Parent ! {result, 1}
  {spread, N} =>
   spawn(testee, self) ! {spread, N-1}
   spawn(testee, self) ! {spread, N-1}
   receive:
    {result, X1} =>
     receive:
      {result, X2} =>
       Parent ! {result, X1 + X2}
```

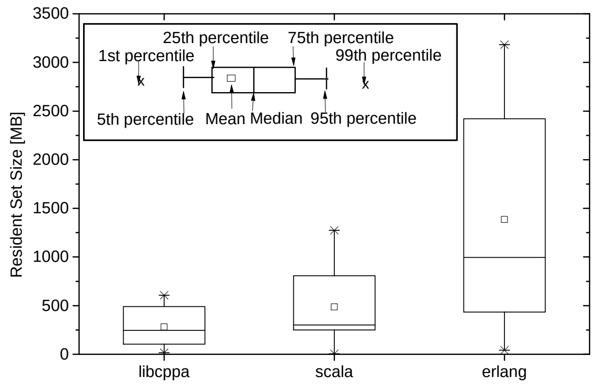
Each actor spawns two additional actors after receiving a `{spread, N}` message unless `N` is 0. In the latter case, the actor sends `{result, 1}`. After spawning two more actors, an actor waits for two result messages, transfers results to its parent, and finishes execution.

This benchmark mainly pressures (a) the heap through many allocations, and (b) the scheduler’s job queue, because actors compute virtually nothing, so the workers are constantly acquiring new tasks. We have run the benchmark with `main(20)`, which creates a total of 2^{20}, i.e., 1,048,576, actors.

Figure 4(a) shows the time to create about a million actors as a function of available CPU cores. `libcppa` is the fastest implementation, halving the runtime of the slowest implementation, i.e., Scala. Erlang’s performance lies midway between the two other implementations. All three implementations reach a global minimum on eight cores. On more than eight cores, three classes can be

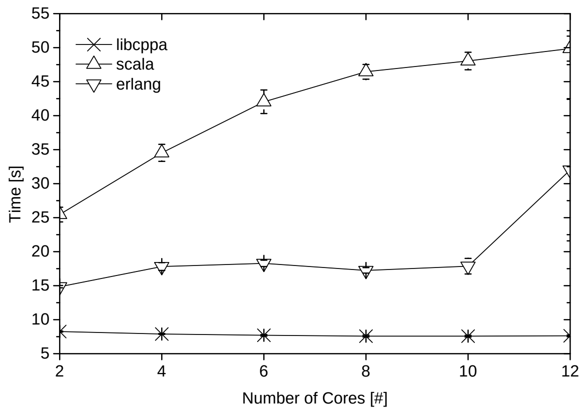


(a) Actor creation time

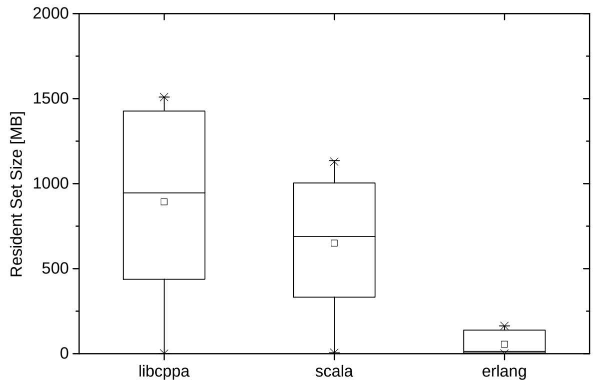


(b) Memory consumption

Figure 4. Actor creation performance for 2^{20} actors



(a) Sending and processing time



(b) Memory consumption

Figure 5. Mailbox performance in N:1 communication scenario

identified: `libcppa` has a steadily increasing curve, Erlang remains almost constant with only a small increase, and Akka exhibits a fluctuating curve with a performance degradation on 10 cores, but it accelerates again on twelve cores.

Figure 4(b) shows the memory consumption during the benchmark. Erlang's actors acquire significantly more memory than all other implementations, peaking above 3 GB of RAM with an average of ≈ 1.5 GB. Scala has an average RAM consumption of 0.5 GB, peaking about 1.5 GB. `libcppa` halves the memory usage of the Scala implementation.

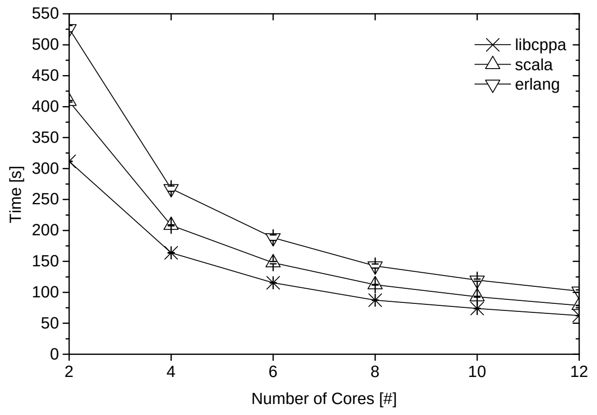
5.3 Mailbox Performance in N:1 Communication Scenario

Our second benchmark measures the mailbox performance in an N:1 communication scenario. We used 20 actors sending 1,000,000 messages each. The minimal runtime of this benchmark is the time the receiving actor needs to process the 20,000,000 messages and the overhead of passing the messages to the mailbox. More hardware concurrency leads to higher synchronization between the sending actors, since the mailbox of the receiving actor acts as a shared resource. Furthermore, the workers in the thread pool are synchronized by a job queue in the implementations using a cooperative scheduling, which can be a concurrency bottleneck as well if actors perform short tasks and often need to be re-scheduled.

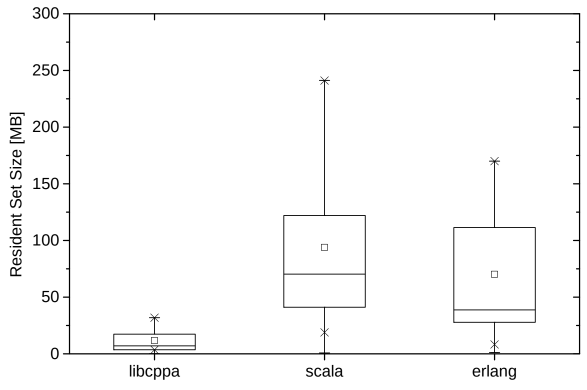
Figure 5(a) visualizes the time needed for the application to send and process the 20,000,000 messages as a function of available CPU cores. The ideal behavior is a decreasing curve, which reaches a global minimum given by the time the receiving actor needs to consume all messages one by one.

The processing overhead increases significantly for Erlang in case of more than 10 cores. `libcppa` attains the best behavior as it reaches its minimum on 4 cores and then remains stable. Akka has a steadily increasing curve up to 10 cores, where it remains stable. The results indicate that our mailbox implementation using the cached stack algorithm (cf., § 4.2.2) as well as our synchronization protocol among worker threads scale very well. In fact, we were not able to evaluate the maximum level of concurrency our implementation could handle in our setup.

Figure 5(b) shows the resident set size during the benchmark execution. In this scenario, a low memory usage hints to a performance bottleneck, as 20 writers should be able to fill a mailbox faster than one single reader is able to drain it. `libcppa` consumes the most memory, because the writers are faster than the reader. Akka consumes $\approx 20\%$ less memory than our C++ implementa-



(a) Sending and processing time



(b) Memory consumption

Figure 6. Performance in a mixed scenario with additional work load

tion, while Erlang needs less than 250 MB of RAM. This outcome is in accord with the performance benchmark. The lower the runtime, the higher the memory consumption.

5.4 Mixed Operations Under Work Load

In this benchmark, we consider a realistic use case including a mixture of operations under severe work load. The benchmark program creates a simple multi-ring topology with a fixed number of actors per ring. A token with an initial value of 1,000 is passed along the ring and decremented each round. A client that receives the token forwards it to its neighbor and terminates whenever the value of the token is 0. Each ring consists of 20 actors and is re-created 20 times. Thus, we continuously create and terminate actors, which process a total of 1,000,000 messages. The following pseudo code illustrates the algorithm.

```
master_loop(Collector, Next):
 receive:
  {token, 0} => // terminate loop
  {token, X} => Next ! {token, X-1}
                master_loop(Collector, Next)

master(Worker, Collector):
 5 times:
  Next = self
  49 times: Next = spawn(chain_link, Next)
  Next ! {token, 1000}
  Worker ! {calc, 28350160440309881}
  master_loop(Collector, Next)
 Collector ! {master_done}

chain_link(Next):
 receive:
  {token, N} => Next ! {token, N}
                if (N > 0) chain_link(Next)

worker(Collector):
 receive:
  {calc, X} => Collector ! {result, fact(X)}
```

We also create one worker per ring that calculates the prime factors of 28,350,160,440,309,881, i.e., 329,545,133 and 86,028,157, to add numerical work load. It is worth noting that this operation is independent of any other actor and does not involve messages. The prime factors are computed, whenever a ring is (re-)created. The calculation requires about two seconds in our loop-based C++ implementation. Our tail-recursive Scala implementation of the prime factorization operates at the same speed, whereas Erlang needs about three seconds.

Each ring consists of 49 `chain_link` actors and one `master`. The `master` re-creates the terminated actors 20 times. Each `master` thus spawns a total of 980 actors. Additionally, there is one message collector and one worker per master. The message collector waits until it receives the result of 400 (20·20) prime factorizations and a *done* message from each master. Overall 19,621 actors are created, but no more than 1,021 actors run concurrently.

Figure 6(a) shows the runtime behaviour as a function of available CPU cores. An ideal characteristic would halve the runtime when doubling the number of cores. All implementations exhibit an almost linear speed-up. For the first time, Akka is faster than Erlang, though it still does not reach the performance of `libcppa`. The performance gap between Erlang and Scala probably results from our previous observation that its prime factorization is about 50% slower.

Figure 6(b) shows the memory consumption during the mixed scenario. `libcppa` has a very constant and thus predictable memory footprint, while using significantly fewer memory than both other implementations. Hence, `libcppa` accounts for the benchmark's characteristics – constant number of actors and messages on average – as memory is released as soon as possible. Both Akka and Erlang exhibit a more unpredictable memory usage, which is not explainable by the benchmark's characteristics. Akka has an average memory consumption of ≈ 75 MB, and peaks about 300 MB. Erlang uses slightly less memory, with an average of ≈ 50 MB, peaking about 175 MB.

5.5 Heterogeneous Computing

Our final benchmark is concerned with heterogeneous computing that includes the use of GPUs. Our first program computes a fixed number of images of the Mandelbrot set using heterogeneous hardware components in a distributed system consisting of 8 worker nodes. The algorithm for calculating the images on a CPU is the same as in our initial evaluation presented in § 2. Our setup consists of a local network connecting nodes at homogeneous 1 Gbit/s links.

Figure 7 shows the runtime behaviour as a function of deployeded processing units. The initial situation, i.e., the zero value on the x-axis, is a setup consisting of 8 worker nodes each using one processor core. As we go along, we strengthen each worker by

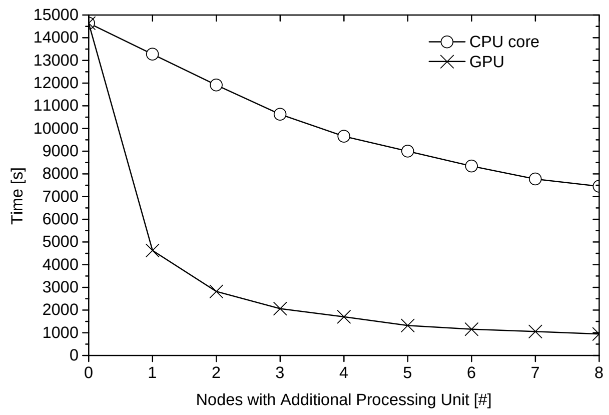


Figure 7. Sending and processing time in a distributed computation using heterogeneous hardware ressources

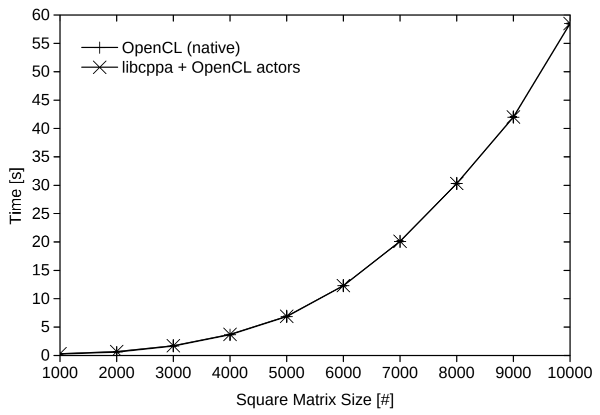


Figure 8. Runtime for using OpenCL natively and wrapped by `libcppa` to multiply two matrices

adding one additional processing unit. In the homogeneous setup (CPU core), we add a second CPU core. In the heterogeneous setup (GPU), we add a GPU instead. Increasing the total number of CPU cores linearly scales down the required runtime by a factor of ≈ 0.9. In the heterogeneous setup, adding only one GPU reduces the runtime to a third. This is, because a GPU outperforms a CPU by an order of magnitude for computationally intense tasks. Doubling the number of GPUs reduces the runtime to $\approx 60\,\%$.

Since `libcppa` wraps the API of OpenCL in order to provide a higher-level abstraction, the efficiency of the mapping is a crucial component for the overall system performance. As shown in § 4.2.6, the `spawn_cl` function compiles the source code of an OpenCL kernel to an actor. Users of `libcppa` need not procure for OpenCL resources managed by the actor system. To benchmark the overhead induced by `libcppa`, we have used an OpenCL program that multiplies two matrices of equal size. Figure 8 shows the runtime results as a function of the matrix size for using OpenCL natively and utilizing the OpenCL abstraction of `libcppa`. The measured runtime overhead added by `libcppa` is no more than 0.07 s with a standard deviation of 0.01 s. Note that this overhead does not only include the wrapping of OpenCL, but also the startup time for `libcppa` and messaging overhead. To estimate the overhead added by libcppa itself, we have used a series of 10,000 runs for a `libcppa`-based program that sends exactly one message to itself and receives it afterwards. The measured average runtime for this simple program is 0.0523435 s. Hence, we can estimate that the runtime overhead of the OpenCL wrapping remains below 0.02 s.

6. Discussion

Object-oriented (OO) programming as found in the dominant programming languages today provides abstraction over data by using encapsulation and dynamic method dispatching. When facing parallelism, the encapsulation achieved by an OO design unfolds. All member functions from shared objects must be protected from parallel access by using locks or semaphores. The caller now has to know implementation details of a member function, thus breaking the abstraction. Furthermore, when composing lock-based, deadlock-free components, the resulting component may very well deadlock [27]. Alternative approaches to OO programming that use message passing rather than method invocation have not been widely adopted. Besides, such approaches are closer to the actor model than to current implementations of the OO paradigm in mainstream languages. Actors can be viewed as higher-order objects providing an abstraction over parallelism. The amount of parallelism in an application is determined by the number of concurrently running actors. Hence, an efficient way to implement applications is to create an actor for each independent task and service. We have shown that creating actors is a lightweight operation (cf. § 5.2) and that communication between actors is built on use an efficient communication layer that scales on multi-processor machines (cf. § 5.3), as well as in distributed systems (cf. § 2).

Some implementations of the actor model tend to be inefficient due to excessive context switching, synchronization and data movement. The event-based actor implementation in our software platform avoids most of the context-switching, since the actor behavior is encoded in message handlers called by resident worker threads. It further allows the workers to bypass the job queue, whenever an actor sends a message to a blocked actor and becomes blocked itself afterwards. The worker thread then chains invocations of event-handlers, which drastically reduces data movement, cache misses and access to the shared job queue.

A central component is our lock-free mailbox implementation (cf., § 4.2.2). It reduces synchronization overhead between actors to a minimum and makes message passing as lightweight as possible. In the N:1 communication scenario, it outperforms all other implementations by a significant margin.

Only the actor creation benchmark revealed minor flaws of our implementation, as the runtime increases on more than eight cores. We want to address this issue by further reducing the amount of RAM each actor occupies. Concurrent memory allocation severely slows down program execution. In our software design, we use one memory allocator per worker thread that allocates and manages memory chunks. We use this custom allocator for actors as well as mailbox elements. This optimization strategy improved the performance of our system by 30–50 %. An even more compact actor implementation would improve the efficiency of our memory management, thus reducing concurrent memory allocation and improving scalability. Such a memory-optimized actor implementation is important to target constraint hardware architectures with only a few KBs of RAM such as embedded systems.

7. Conclusion and Outlook

This paper presented a software platform for development of native applications running in distributed, heterogeneous environments. Our software platform offers (1) a message-oriented API based on the actor model for developing high-performance, low-latency applications in C++, (2) an automated approach to generate a message passing interface for heterogeneous hardware components such as graphics cards, and (3) a common, high-level message exchange layer for distributed applications. To best of our knowledge, there exists no messaging library for a native programming language offering a comparable high-level abstraction for message passing. Previous attempts to implement the actor model for C++ either lack important characteristics of an actor system, e.g., network transparency (Theron[2]), or do not provide a sufficient level of abstraction to communicate to actors without knowledge of the class definition of the receiving actor (ACT++[18]), or are bound to vendors (MS Casablanca [20]) for intentionally limited use. In addition to traditional features of an actor system, we also provide a publish/-subscribe layer, to allow for loosely coupled distribution and rendezvous protocols [7].

We have contributed our implementation to the C++ community [6] and thoroughly evaluated our software platform (cf. § 5). In our platform, actor communication is distribution-transparent and, as measurements indicate, sufficiently fast. Our cooperative scheduling is scalable and competes with mature implementations of the actor model. Since our software platform is implemented in a native programming language without garbage collector, it uses fewer memory than comparable approaches based on virtual machines.

Compared to low-level message passing implementations such as OpenMPI, the tuple-based, dynamic nature of actor communication causes a higher transmission and message handling overhead. Still, the communication layer offered by our software platform is on par with low-level message passing libraries such as OpenMPI.

Our future work currently focuses on supporting embedded hardware systems. This development aims to provide a high-level software development platform for distributed systems running on strictly constraint hardware such as wireless sensor nodes. Our software platform would allow for local testing of software components, i.e., actors, before distributing the application by binding each component to an embedded system.

8. Acknowledgements

The authors would like to thank the iNET working group for vivid discussions and inspiring suggestions as well as the anonymous reviewers for their constructive and insightful feedback. Funding by the German Federal Ministry of Education and Research within the projects SAFEST and ScaleCast is gratefully acknowledged.

References

[1] G. Agha. Actors: A Model Of Concurrent Computation In Distributed Systems. Technical Report 844, MIT, Cambridge, MA, USA, 1986.

[2] G. Agha, I. A. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. In *Proceedings of CONCUR*, volume 630 of *LNCS*, pages 565–579, Heidelberg, 1992. Springer-Verlag.

[3] J. Armstrong. Erlang - A Survey of the Language and its Industrial Applications. In *Proceedings of the symposium on industrial applications of Prolog (INAP96)*, pages 16–18. Hino, October 1996.

[4] J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, KTH, Sweden, 2003.

[5] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud Computing for Everyone. In *Proc. of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 16:1–16:14, New York, NY, USA, 2011. ACM.

[6] D. Charousset and T. C. Schmidt. libcppa - Designing an Actor Semantic for C++11. In *Proc. of C++Now*, May 2013.

[7] D. Charousset, S. Meiling, T. C. Schmidt, and M. Wählisch. A Middleware for Transparent Group Communication of Globally Distributed Actors. In *Middleware Posters 2011*, New York, USA, Dec. 2011. ACM, DL.

[2] See `http://www.theron-library.com`

[8] J. O. Coplien. Curiously Recurring Template Patterns. *C++ Report*, 7:24–27, February 1995.

[9] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal Verification of a Practical Lock-Free Queue Algorithm. In *FORTE*, volume 3235, pages 97–114. Springer, September 2004.

[10] M. Fouquet, H. Niedermayer, and G. Carle. Cloud Computing for the Masses. In *Proc. of the 1st ACM Workshop on User-provided networking: challenges and opportunities*, U-NET '09, pages 31–36, New York, NY, USA, 2009. ACM.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Comput.*, 22(6):789–828, Sept. 1996.

[12] P. Haller and M. Odersky. Scala Actors: Unifying Thread-Based and Event-Based Programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[13] P. B. Hansen. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973. ISBN 0-13-637843-9.

[14] M. Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[15] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th ISCA*, pages 289–300, New York, NY, USA, May 1993. ACM.

[16] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd IJCAI*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[17] I.B.M. Corporation. IBM System/370 Extended Architecture, Principles of Operation. Technical Report SA22-7085, IBM, 1983.

[18] D. G. Kafura and K. H. Lee. Act++: Building a concurrent c++ with actors. *J. Object Oriented Program.*, 3(1):25–37, Apr. 1990.

[19] S. Meyers and A. Alexandrescu. C++ and the Perils of Double-Checked Locking. *Dr. Dobb's Journal*, July 2004.

[20] Microsoft. Casablanca. `http://msdn.microsoft.com/en-us/devlabs/casablanca.aspx`, 2012.

[21] D. G. Murray, E. Yoneki, J. Crowcroft, and S. Hand. The Case for Crowd Computing. In *MobiHeld '10: Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds*, pages 39–44, New York, NY, USA, 2010.

[22] J. H. Nyström, P. W. Trinder, and D. J. King. Evaluating Distributed Functional Languages for Telecommunications Software. In *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, ERLANG '03, pages 1–7, New York, NY, USA, 2003. ACM.

[23] M. Rettig. Retlang. `code.google.com/p/retlang`, December 2010.

[24] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the fourteenth annual ACM symposium on PODC*, pages 204–213, New York, NY, USA, 1995. ACM.

[25] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd ECOOP*, volume 5142 of *LNCS*, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12 (3):66–73, May 2010.

[27] H. Sutter and J. Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, Sept. 2005.

[28] TIOBE Software BV. Programming Community Index. `www.tiobe.com`, April 2012.

[29] J. Torrellas, H. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Trans. Comput.*, 43(6):651–663, June 1994.

[30] Typesafe Inc. Akka. `akka.io`, March 2012.

[31] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, December 2001.

[32] T. Veldhuizen. Expression Templates. *C++ Report*, 7:26–31, 1995.