Annales Mathematicae et Informaticae
53 (2021) pp. 271–282
DOI: https://doi.org/10.33039/ami.2021.03.006
URL: https://ami.uni-eszterhazy.hu

Exploiting the structure of communication in actor systems

Krisztián Schäffer[a], **Csaba István Sidló**[b]

[a]Monotic Mo. Kft., Budapest, Hungary
krisztian.schaffer@monotic.com

[b]Institute for Computer Science and Control (SZTAKI), Budapest, Hungary
sidlo@sztaki.hu

Submitted: December 23, 2020
Accepted: March 8, 2021
Published online: May 18, 2021

Abstract

We propose a novel algorithm for minimizing communication costs of multi-threaded and distributed actor systems, to gain performance advantage by dynamically adapting to the structure of actor communication. We provide an implementation in Circo, an open source actor system, and show promising experimental results.

Keywords: Actor model, concurrent systems

1. Introduction and related work

Actor-based concurrency models [1] have been used for decades for scalable distributed applications [11]. Actors – the primitives of concurrency – encapsulate their state, communicate through asynchronous messaging and form arbitrary topological relations.

Various frameworks and languages permit actor programming, including Akka [15], CAF [7] and Pony [10]. Applications include banking and telecom transaction processing, complex event stream processing and large-scale analytical pipelines. The concurrency model of microservice architectures [8] corresponds with the actor model, and actor frameworks can be applied directly in cloud environments (e.g. Orleans [4]). Driven by the popularity of cloud and Internet of Things (IoT)

solutions and the stagnating performance of single CPU cores, the last few years has seen an increased interest in actor systems. We believe that actor systems also have a great potential for artificial intelligence, by providing an efficient tool to incorporate sparsity into deep learning.

1.1. Why actors?

Programs built using other programming models – especially the synchronous ones – may be easier to reason about, but the actor model allows unlimited scaling and a variety of performance optimizations thanks to a few key properties:

1. *No shared state:* An actor can access only its own state directly, and everything else must be done through messaging. Shared state is an abstraction famous for introducing hard to find bugs called data races in concurrent programs. Actor programming does not expose the programmer to the risks of shared memory, leaving shared memory to automatic performance optimizations.
2. *No global synchronization mechanism included:* Synchronization must be implemented on the actor level, using the fact that message processing of a single actor is serializable.
3. *Location transparency:* The act of sending a message does not depend on the location of the target actor – sending messages within a machine is the same as between machines.

Global synchronization performance degrades as the physical diameter of the system grows, because information cannot travel faster than light. Similarly, providing the illusion of synchronous shared state – which does not exists in reality - is only possible with introducing a latency proportional to the diameter of the subsystem containing the state. Not having these features allows the actor model to scale arbitrarily without performance loss. The third property, location transparency, allows the execution environment to optimize actor placement and message passing during run-time without actors noticing it.

1.2. Communication complexity

Communication is a common performance bottleneck of distributed systems, even on single-node multi-core systems, where shared-memory communication between cores works well, but brings in significant latency.

Communication is layered in modern hardware: network is slower than shared-memory which is slower than in-thread (cached) data passing. This layeredness of technology is a result of physical and technological constraints, namely the speed of light, maximal density of hardware elements, and manufacturing costs. It is reasonable to think that these constraints and the technology layers will not disappear soon. Even if the layers merge, messaging latency remains dependent on physical distance, because information cannot travel faster than light. Communi-

cation therefore will remain a performance limiting factor of distributed systems for long.

Large scale data processing systems apply data locality to minimize the cost of communication – bringing computation to the data [6]. Disk-based data locality was a key success factor of MapReduce, but as network technology outpaced local storage speed, memory locality became the primary goal. Data processing frameworks are now often aware of locality, with regard to NUMA (Non-Uniform Memory Access) patterns [13]. Actor systems are also employing techniques to deal with locality in non-uniform shared memory: a locality-guided scheduler for CAF was published in [14], and locality-aware work stealing scheduler methods was studied in [2].

The main goal of this paper is to provide a general method to reduce communication overhead in distributed systems. We formulate a solution in the context of the actor model: the decentralized "infoton optimization" algorithm is presented, which explores and exploits the structure of communication to minimize communication cost by co-locating actors during run-time. The computational cost of this algorithm is proportional to the number of actor messages.

2. Infoton optimization

To reduce communication costs during execution of an actor system, frequently communicating actors are to be moved to a common, or at least to nearby locations – e.g. to the same NUMA location, computer or data center.

Infoton optimization is a physics-inspired model, essentially a decentralized, scalable version of force-directed graph drawing [12] – a physical system of bodies with cohesive forces, where the energy of the system is to be minimized. Infoton optimization maps intensity of actor communication to forces of the physical system and approximates the behavior of the system in a way that needs no central coordination.

Actors and schedulers (threads executing actor code) are mapped to 3D Euclidean space: The main idea is that distance approximates communication cost, and actors move towards their communication partners to minimize communication cost.

Euclidean space is chosen on purpose as the model of the physical universe, where communication cost often depends on physical distance – even multicore CPUs evolve to be 3D structures [3]. However, as network and other communication costs don't always match the strict Euclidean properties, other spaces might also be investigated.

Schedulers are embedded in a way that their distance represents communication overhead, either by static positioning, or by using network coordinates. Actors move in the space during optimization and are continuously migrated to the nearest scheduler.

2.1. Assumptions

Infoton optimization assumes the following properties of the actor system:

1. *Actors are significantly more numerous than threads.*
2. *Actor communication is structured:* Only a small, slowly changing portion of possible actor connections is used.
3. *Actors can be migrated:* Computational load with actors can be moved between schedulers, affecting future communication cost.

2.2. What is an infoton

The infoton is the quantum of actor forces, a force-carrying particle – like photons in physics – that:

1. Is coming from a *source location.*
2. Carries a positive scalar value called *energy.*
3. Has a *sign.*

2.3. Infoton action

When the infoton acts on an actor, it either pulls or pushes the actor toward/away from the source location of the infoton. The direction of the actor movement depends on the *sign* (positive pulls), while the distance is proportional to the *energy* of the infoton.

Actors have no inertia in this model, they only move when infotons act on them. This way inactive actors introduce no computational overhead. The physical analogy is that actors move in thick fluid.

2.4. The first force

We define two major forces of infoton optimization. The first force of infoton optimization brings communicating actors toward one another.

1. An infoton is attached to every message passed between actors, holding the *position of the source actor* and a unit of *energy* with positive sign.
2. When the message arrives at its destination actor, the infoton attached to it acts on that actor, *pulling it towards the source of the message.*

2.5. The second force

Another force spreads actors in the segment of the space near schedulers, avoiding all concentrating around a single point:

1. When a message arrives, the scheduler that executes the target actor creates a new "scheduler infoton", with itself as source.
2. Scheduler infotons either pull or push actors toward or away from the scheduler, depending on the load of the scheduler.

3. Implementation details

We have added an experimental implementation to the Circo [9] actor system (where the main author and maintainer is the main author of this paper). Circo is written in Julia [5], a dynamically typed, garbage collected general-purpose language designed for numerical computing.

Unlike most contemporary actor systems, Circo implements multi-threading by running several single-threaded schedulers that communicate through shared memory and form the "host cluster". Similarly, distribution of work between hosts is done in a separated cluster, which we call "the" Cluster, because we think that the Actors stick to schedulers by default, but can migrate between them by using the migration service.

For simplicity the current implementation of infoton optimization assumes that communication overhead between any pair of schedulers is fixed, thus schedulers are statically positioned. This, however, can be extended to be dynamically adjusted.

The algorithm can be customized with the following parameters:

1. I – A proportionality constant of actor forces, similar to the G gravitational constant in physics. It connects the energy of the acting infoton to the length of movement caused by the action. Higher values cause more intense actor movement.
2. *TARGET_DISTANCE* – Force-directed graph drawing algorithms often use repulsive forces between every pair of nodes to avoid the concentration of nodes. The second force of infoton optimization has a similar goal, but we have found that the algorithm is more stable with a quirk that approximates a hidden repelling force acting only at low distances: When the source position of a pulling infoton is closer to the target actor than *TARGET_DISTANCE*, its effect is extinguished.
3. *SCHEDULER_TARGET_LOAD* – We define the load of a scheduler as the total number of messages waiting to be processed. This parameter sets the load that every scheduler tries to maintain independently. Scheduler infotons emitted by a scheduler will pull actors when its load is lower and push when higher.
4. *SCHEDULER_LOAD_FORCE_STRENGTH* – Proportionality constant of scheduler infoton energy.

Following is the Julia code that calculates the movement of an actor caused by an infoton acting on it (error handling is not shown):

```
function Circo.apply_infoton(targetactor, infoton)
    diff = infoton.sourcepos - targetactor.core.pos
    difflen = norm(diff)
    energy = infoton.energy
    if energy > 0 && difflen < conf[].TARGET_DISTANCE
        return nothing
    end
    stepvect = diff / difflen * energy * conf[].I
```

```
    targetactor.core.pos += stepvect
    return nothing
end
```

The code to generate the "scheduler infoton" when delivering a message:

```
function Circo.scheduler_infoton(scheduler, actor)
    dist = norm(scheduler.pos - actor.core.pos)
    loaddiff = Float64(conf[].SCHEDULER_TARGET_LOAD - length(scheduler.msgqueue))
    if loaddiff == 0.0 # Nothing to do at target load
        return Infoton(scheduler.pos, 0.0)
    end
    energy = sign(loaddiff) *
             log(abs(loaddiff)) *
             conf[].SCHEDULER_LOAD_FORCE_STRENGTH
    return Infoton(scheduler.pos, energy)
end
```

Although Circo supports multi-threaded and distributed settings, for easier experimentation we have created a simulation environment[1] that starts several schedulers on the same thread and allows changing of optimization parameters during run-time.

4. Experiments

We have conducted experiments with two actor programs:

1. *Linked List:* Generates a linked list of actors, each storing a single scalar, then runs reduce (sum) operations on the list concurrently. When an operation finishes, immediately starts a new one, maintaining 100 concurrent operations.
2. *Search Tree:* Generates a binary search tree of actors, leafs hold 1000 scalars, inner nodes contain only a split value and addresses of two children. Fills the tree with random data and runs search operations concurrently (during and after filling, maintaining 500 concurrent searches).

In both cases a coordinator actor manages the creation of the data structure and sends the reduce/search operations to it. Results are sent back to the coordinator, so the computing graphs are cyclic: A single cycle containing every actor of the linked list, and a unique cycle for every leaf of the search tree.

We have introduced "domain knowledge" to the search tree through two simple actor behaviors, improving performance. We call these behaviors domain specific, because they reflect information about the structure of the actor system (that it is a tree). First, the coordinator actor goes back to the fixed position $(-10, 0, 0)$ every time it receives a search response. This helps stabilizing the tree layout. Second, tree nodes periodically send a negatively signed infoton to their siblings in order to

[1] To reproduce the experiments, open `https://github.com/Circo-dev/ExploreInfotonOpt`.

repel each other. This is intended to open up the tree, making easier for siblings to separate.

Actors are continuously moving and occasionally changing schedulers when they get closer to another one. Messages are considered local if source and destination actors run on the same scheduler. As the direct indicator of optimization success we have measured the ratio of the number of local messages to the total number of messages.

To demonstrate that it is possible to find a single set of parameters for which infoton optimization yields good results for a wide variety of actor programs, we have run differently sized versions of the two programs with the same fixed infoton optimization parameters, selected manually:

- I = 0.2,
- TARGET_DISTANCE = 200.0,
- SCHEDULER_TARGET_LOAD = 13,
- SCHEDULER_LOAD_FORCE_STRENGTH = 0.02

Six schedulers were used, positioned at face-centers of a cube: $(-1000, 0, 0)$, $(1000, 0, 0)$, $(0, -1000, 0)$, $(0, 1000, 0)$, $(0, 0, -1000)$, $(0, 0, 1000)$.

Figures in this paper are screenshots of the Circo tool "Camera Diserta", used to monitor and validate actor layout. Grey lines are local, orange lines are non-local connections between actors. Schedulers are drawn as blue cubes, test coordinator as a red sphere.

Figure 1 illustrates the layout of the linked list program with 200 (71%), 500 (83%), 1000 (85%), 2000 (89%), 4000 (89%) and 8000 (89%) (row major order) list item actors. Percents in parentheses are local message ratios of the last 10 seconds before taking the screenshots. (When actors are distributed randomly between six schedulers, local message ratio is 1/6 (17%).)

Figure 2 illustrates the layout of the search tree program with 62 (51%), 126 (57%), 254 (64%), 506 (66%), 1018 (70%), 2028 (73%), 4046 (74%) and 8080 (76%) (row major order) tree node actors. Percents in parentheses are local message ratios of the last 10 seconds before taking the screenshots. Connections from leafs to the coordinator are not drawn.

Figure 3 illustrates the layout of the search tree without the domain-specific behaviors, with 62 (45%), 96 (46%), 254 (51%), 510 (50%), 1017 (51%), 2004 (56%) tree node actors. These layouts are not stable, they are slowly and continuously restructuring while maintaining high local message rate. Note that connections near the leafs have much less message traffic than near the root, so the optimization is still successful despite the high amount of non-local connections.

Infoton optimization radically improved message locality in all three experiments, reducing inter-scheduler communication by 43–87% compared to the random placement baseline. Figure 4 illustrates this by showing local message ratios achieved after optimization.

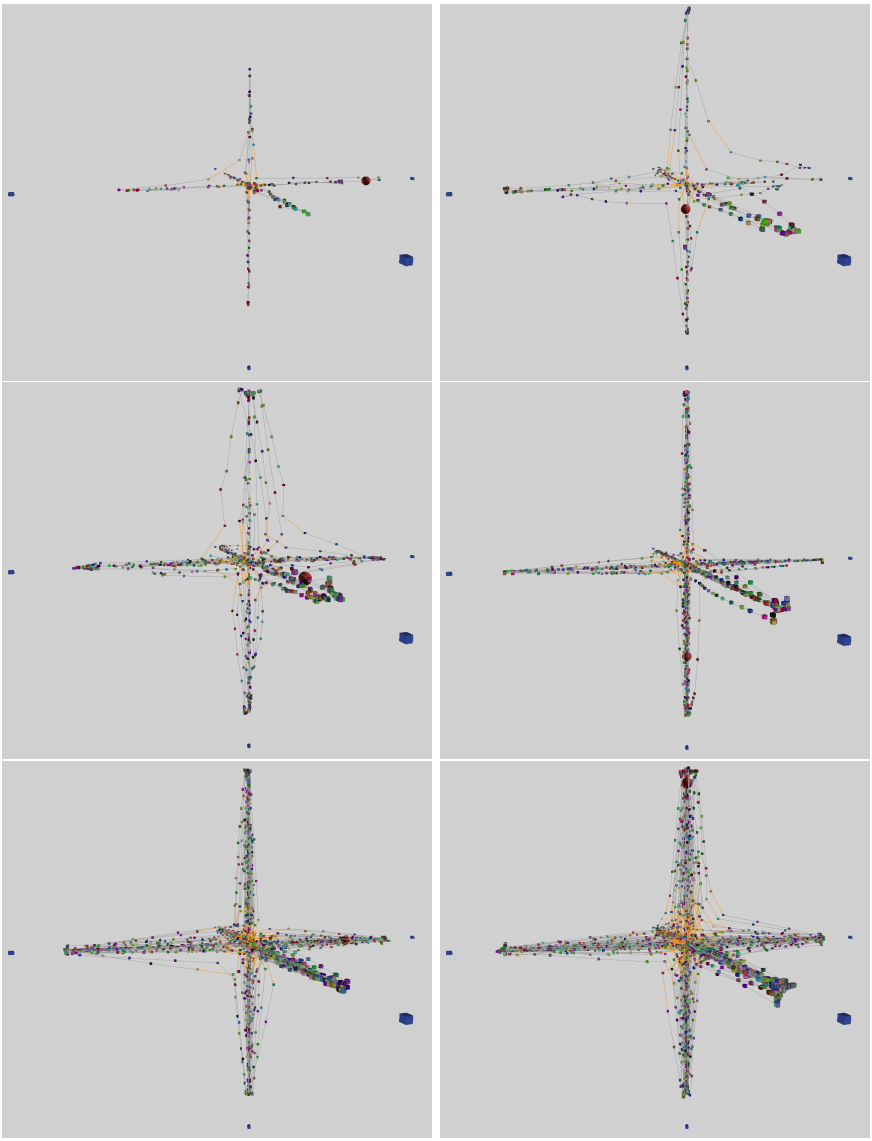

Figure 1. Optimized layouts of a linked list of 200, 500, 1000, 2000, 4000 and 8000 actors on 6 schedulers.

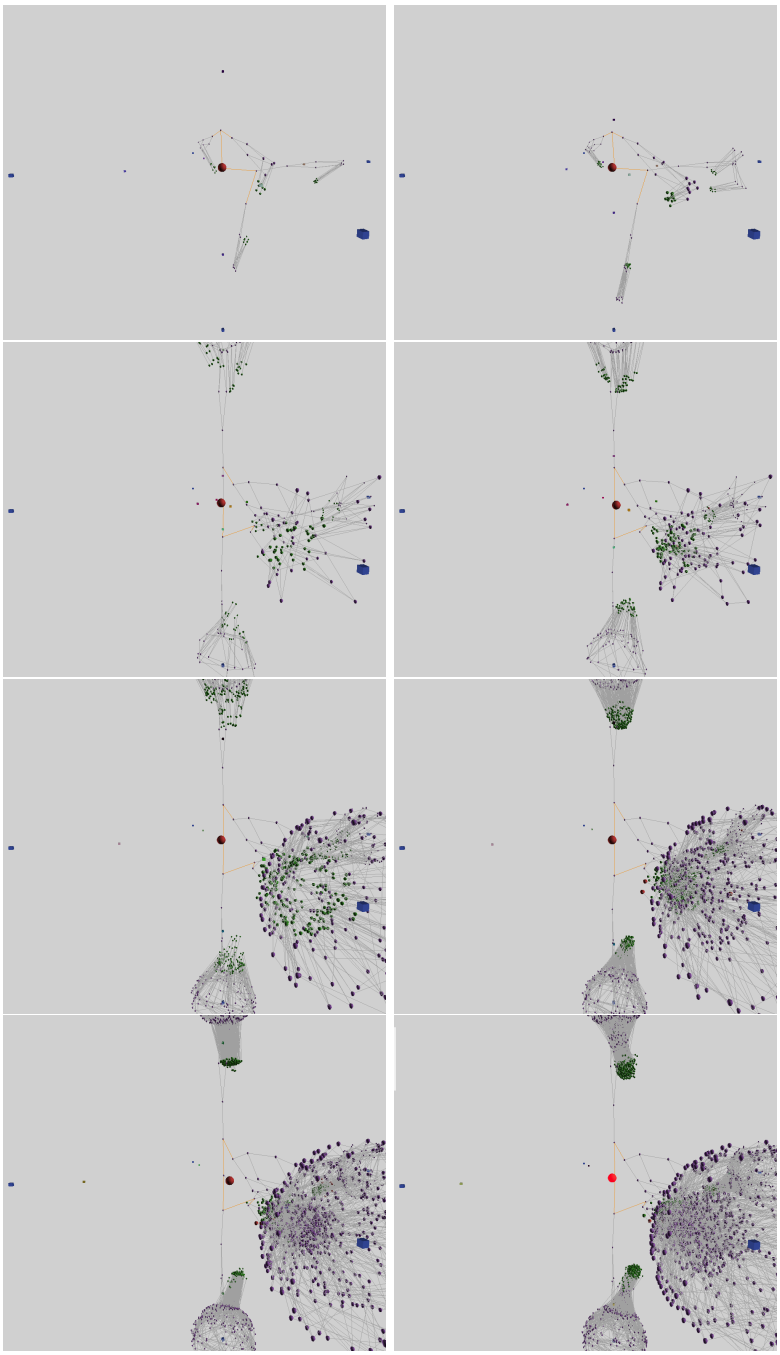

Figure 2. Optimized layouts of a binary tree built from 62, 126, 254, 506, 1018, 2028, 4046 and 8080 actors on 6 schedulers.

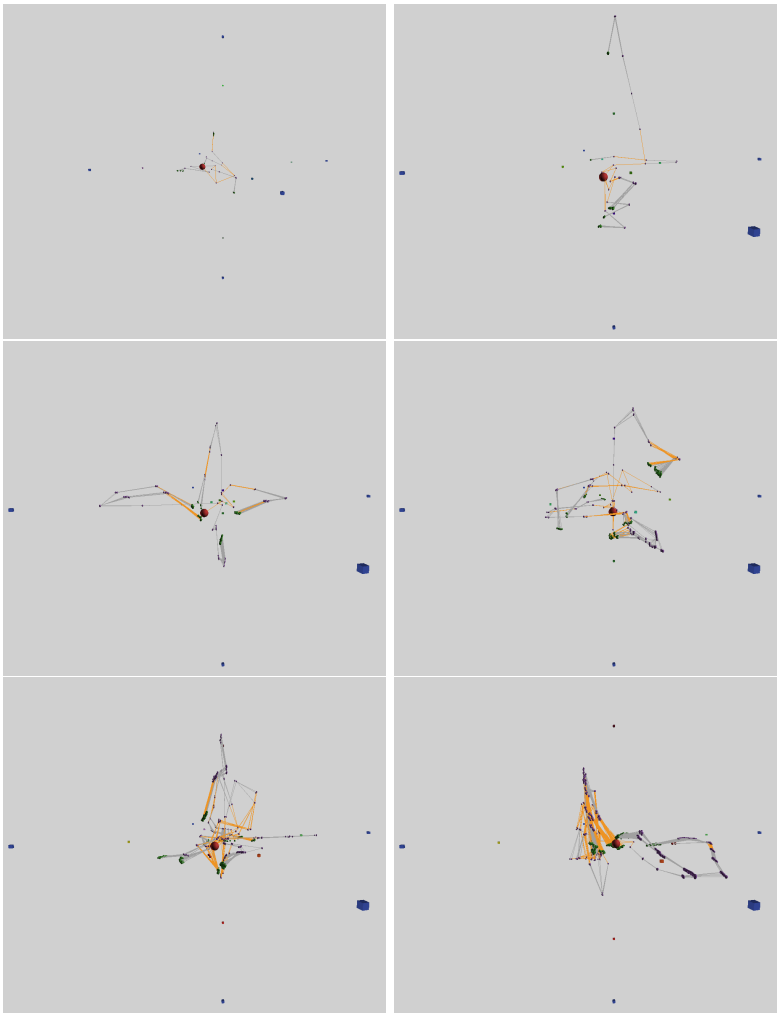

Figure 3. Optimized layouts of a binary tree without the domain-specific behaviors, built from 62, 96, 254, 510, 1017, 2004 actors on 6 schedulers.

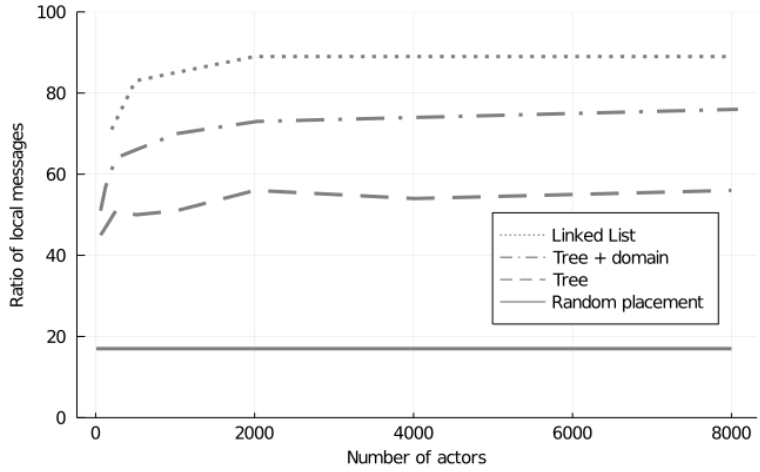

Figure 4. Local message ratios achieved at different program sizes on 6 schedulers, and random actor placement as baseline.

5. Conclusions and future work

We have introduced infoton optimization, and demonstrated in a limited scenario that it is capable of distributing computational load of actor systems while optimizing message locality. The algorithm is decentralized and its cost is proportional to the number of messages.

The algorithm has several parameters that need to be tuned manually. Manual tuning of large decentralized systems may not always be feasible, so future work should focus on meta-optimization or elimination of these parameters.

One of our examples introduces domain-specific constraints on how actors behave, which improves the efficiency of the optimization significantly. However, the optimization works well without these domain-specific behaviours too. This shows that the algorithm is easily customizable with (application-specific) domain knowledge, and for some actor programs such customization may result in significant performance gain.

In the simple version of the algorithm discussed in this paper, actors behave uniformly when infotons act on them. Introducing "mass" or "size" properties of actors to reflect the cost of migration is however a promising extension of the algorithm.

Several further aspects of infoton optimization are to be clarified and detailed as future work. For example, convergence criteria of infoton optimization and optimality of the results are studied in the context of stochastic optimization. Detailed benchmark experiments are also being performed, comparing common actor systems with Circo.

References

[1] G. Agha: *Actors: A Model of Concurrent Computation in Distributed Systems*, Cambridge, MA, USA: MIT Press, 1986, isbn: 0262010925.

[2] S. Barghi, M. Karsten: *Work-Stealing, Locality-Aware Actor Scheduling*, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018, pp. 484–494, doi: https://doi.org/10.1109/IPDPS.2018.00058.

[3] A. Ben Abdallah: *3D Integration Technology for Multicore Systems On-Chip*, in: Advanced Multicore Systems-On-Chip: Architecture, On-Chip Network, Design, Singapore: Springer Singapore, 2017, pp. 175–199, isbn: 978-981-10-6092-2, doi: https://doi.org/10.1007/978-981-10-6092-2_6.

[4] P. A. Bernstein, S. Bykov: *Developing Cloud Services Using the Orleans Virtual Actor Model*, IEEE Internet Computing 20.5 (2016), pp. 71–75, doi: https://doi.org/10.1109/MIC.2016.108.

[5] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah: *Julia: A fresh approach to numerical computing*, SIAM review 59.1 (2017), pp. 65–98.

[6] S. Bonner, I. Kureshi, J. Brennan, G. Theodoropoulos: *Chapter 14. Exploring the Evolution of Big Data Technologies*, in: Dec. 2017, isbn: 9780128054673, doi: https://doi.org/10.1016/B978-0-12-805467-3.00014-4.

[7] D. Charousset, R. Hiesgen, T. C. Schmidt: *Revisiting Actor Programming in C++*, CoRR abs/1505.07368 (2015), arXiv: 1505.07368.

[8] Chris Richardson: *What are microservices?*, https://microservices.io/, [Accessed: 15 November 2020].

[9] *Circo: A fast, scalable and extensible actor system.* https://github.com/Circo-dev/Circo, [Accessed: 22 December 2020].

[10] S. Clebsch, S. Drossopoulou, S. Blessing, A. McNeil: *Deny Capabilities for Safe, Fast Actors*, in: AGERE 2015, Association for Computing Machinery, 2015, pp. 1–12, doi: https://doi.org/10.1145/2824815.2824816.

[11] J. De Koster, T. Van Cutsem, W. De Meuter: *43 years of actors: a taxonomy of actor models and their key properties*, in: Oct. 2016, pp. 31–40, doi: https://doi.org/10.1145/3001886.3001890.

[12] T. M. Fruchterman, E. M. Reingold: *Graph drawing by force-directed placement*, Software: Practice and experience 21.11 (1991), pp. 1129–1164.

[13] T. Li, Y. Ren, D. Yu, S. Jin: *Analysis of NUMA effects in modern multicore systems for the design of high-performance data transfer applications*, Future Generation Computer Systems 74 (2017), pp. 41–50, issn: 0167-739X, doi: https://doi.org/10.1016/j.future.2017.04.001.

[14] S. Wölke, R. Hiesgen, D. Charousset, T. C. Schmidt: *Locality-Guided Scheduling in CAF*, in: Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2017, Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 11–20, isbn: 9781450355162, doi: https://doi.org/10.1145/3141834.3141836.

[15] D. Wyatt: *Akka Concurrency*, Sunnyvale, CA, USA: Artima Incorporation, 2013, isbn: 0981531660.