Locality-Guided Scheduling in CAF

Sebastian Wölke
Dept. Computer Science
Hamburg University of Applied Sciences
Germany
Sebastian.woelke@haw-hamburg.de

Raphael Hiesgen
Dept. Computer Science
Hamburg University of Applied Sciences
Germany
raphael.hiesgen@haw-hamburg.de

Dominik Charousset
Dept. Computer Science
Hamburg University of Applied Sciences
Germany
dominik.charousset@haw-hamburg.de

Thomas C. Schmidt
Dept. Computer Science
Hamburg University of Applied Sciences
Germany
t.schmidt@haw-hamburg.de

Abstract

The C++ Actor Framework (CAF) was designed for using multiple, exchangeable schedulers with a default choice of random work stealing (RWS) for load-balancing. RWS is excellently scalable, and by choosing a random victim scheduling is kept simple with minimal information required. On the downside, it ignores data locality and misses opportunities to improve the application performance.

In this paper, we contribute a locality-guided scheduling that exploits knowledge about the host system to adapt runtime deployment and thereby improves the performance of actor based applications. We implement and thoroughly analyze a CAF scheduler which considers the trade-off between *communication locality* and *execution locality*. The former describes the locality of communicating actors, while the latter the locality between a worker, which executes an actor, and the location of its data. Extensive performance evaluations show a performance gain for data intensive application of up to 25% on a 64 core NUMA machine.

CCS Concepts • **Computing methodologies → Concurrent programming languages**; • **Software and its engineering → Scheduling**; **Software performance**; Multiprocessing / multiprogramming / multitasking; • **Information systems →** *Data layout*;

Keywords Actor Model, Scheduling, Data Locality, NUMA

ACM Reference Format:
Sebastian Wölke, Raphael Hiesgen, Dominik Charousset, and Thomas C. Schmidt. 2017. Locality-Guided Scheduling in CAF. In *Proceedings of 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3141834.3141836

1 Introduction

Concurrent programming becomes continuously more important as the number of cores per CPUs increases while single core performance stagnates. Fully taking advantage of multicore systems requires special care from programmers to coordinate computations across multiple processing units. A powerful computation model that overcomes these obstacles and addresses concurrency problems like low level race conditions and deadlocks is the actor model [11]. Actors are lightweight, independent, and isolated entities that solely interact via asynchronous message passing and allow for scaling applications to many cores.

The C++ Actor Framework (CAF) [5, 6] is an implementation of the actor model. Written in the C++11 standard, the framework provides native program execution as well as a high level of abstraction for writing concurrent and distributed applications with a focus on scalability. CAF is designed with a modular architecture that allows developers to extend or exchange components such as the scheduler.

The scheduler of an actor system is a performance critical component. Leaving it ill-configured or choosing an unfit scheduling strategy can slow down applications when CPUs are left idle and work is not balanced across the available cores efficiently. CAF uses random work stealing (RWS) [4] by default, a decentralized scheduling approach with excellent scalability. An RWS scheduler deploys a number of workers, each of which owns a job queue and when it drains steals from a random victim.

The memory architecture of modern processors is structured hierarchically. This leaves CPUs with inhomogeneous performance characteristics depending on the memory region they access. Multiple levels of caches and a non-uniform memory access (NUMA) architecture are introduced to compensate for these conditions. Extending a scheduler to take data locality into account can improve the performance of applications that utilize heterogeneous memory architectures.

In this work, we present a locality-guided scheduling (LGS) approach that exploits knowledge about the memory architecture to improve the performance of actor-based applications. For this purpose, LGS considers *communication locality* (CL) [15], the locality of communicating actors, and *execution locality* (EL) [17], the locality between a worker and the data

AGERE'17, October 23, 2017, Vancouver, Canada
2017. ACM ISBN 978-1-4503-5516-2/17/10…$15.00
https://doi.org/10.1145/3141834.3141836

of the actor it executes. Locality describes the arrangement of entities and data over CPUs, caches, and memory banks. We devise a combination of *weighted work stealing* and *actor pinning* that enables LGS to find a trade-off between the two.

The remainder of this paper is structured as follows. Section 2 discusses scheduling challenges and design constrains along with related work. Section 3 describes locality-guided scheduling in detail, which is evaluated in Section 4. Finally, Section 5 concludes and gives an outlook to future work.

2 Challenges of a Locality-Aware Scheduling and Related Work

Concurrent software consists of chunks that can be executed in parallel. In the actor model, these chunks construct a dynamic communication network consisting of a varying number of actors. A scheduler assigns these actors (work items) to a pre-allocated number of workers distributed across processing units (PUs). Since current processing hardware runs much faster than data can be retrieved from main memory, the performance of an actor critically depends on the amount of data it processes and its memory access pattern. To reduce waiting times for data delivery, modern hardware is equipped with multiple levels of caches which are preloaded and temporally store data for future use. The first level (L1) caches are often tightly coupled to specific PUs and on par in speed but small. Additional cache levels are larger, slower and often shared between a subset of PUs.

Modern multicore systems use multiple memory controllers in a non-uniform memory access (NUMA) architecture [10]. Bundling PUs together with memory banks into NUMA nodes allows commodity hardware to scale linearly with the number of available PUs as long as the executed software threads work on distinct memory regions. NUMA-nodes are connected via links such that each PU can transparently access the memory of other nodes—although with varying access times. The more hops are required to access another NUMA node, the slower memory access becomes. Load on links along the path can further slow down memory access time. An actor framework must become aware of memory heterogeneity and adapt to access locality and caching to efficiently cope with these hardware characteristics [8].

2.1 Scheduling Constraints

A scheduler for a user-space actor system such as implemented in CAF—the C++ Actor Framework [5, 6]—is constrained in multiple dimensions and requires careful consideration of trade-offs between conflicting goals.

When scheduling actors in such an environment the runtime system has no a priori knowledge of the application behavior and thus must implement an approach that performs well for a large number of versatile use cases for actors.

The general aim of scheduling optimizations is to minimize the overall runtime of an application which is called makespan in the context of scheduling. Finding the best scheduling decision is a well-known optimization problem called the *Job Shop Scheduling Problem* [2]. It describes the challenge assigning a number of work items of varying execution times to a number of PUs. Finding the optimal solution is an NP-hard, offline problem. Information such as the number of work items, their dependencies, and individual execution times are provided upfront when solving an offline problem. In contrast, the CAF scheduler attempts to solve an online problem where work items are generated dynamically and no estimates for individual execution times exist.

Prioritizing performance-critical work items such as items that have many dependencies to future work is a strategy to reduce the makespan [21]. This requires knowledge about the application behavior ahead of time. While this would be possible in the presence of a deterministic execution model, actor systems are non-deterministic [14]. This renders the approach based on makespan impractical.

Work items can either be scheduled in a preemptive or cooperative fashion. A preemptive scheduler can interrupt its work items during execution, e.g., to reschedule them after a defined period of time or when priorities change. This can be used to protect work items from starvation or to enable fair sharing of CPU time. In contrast, a cooperative scheduler waits until work items voluntarily yield control. This usually reduces the number of context switches and causes less overhead than a preemptive scheduler at the price of possibly unfair resource utilization. A preemptive scheduler can interrupt work items either on the operating system level or within a virtual machine in user space. Designed as a native library that runs in user space, CAF remains restricted to cooperative scheduling.

2.2 Problems of Locality

A scheduler can optimize the communication effort between actors, called communication locality (CL) [15], or the efforts of a PU of accessing the data by its executing actor, known as execution locality (EL) [17]. Note that both locality aspects are not restricted to actor parallelism but exist for task and thread parallelism as well. Here, CL occurs indirectly, e.g., when passing on a result from one task to the next one. CL influences the performance of inter-actor message exchange. In the best case, communication partners are executed on the same PU where they may share data stored in L1-cache. In the worst case, the actors are located at different NUMA-nodes and data must be accessed remotely.

Communication with memory-mapped I/O devices is affected by CL similarly to actor-to-actor communication as devices are connected to specific NUMA-nodes. Exchanging many or large messages between two tightly-coupled actors performs best when scheduling both actors to the same PU.

EL quantifies the time required to access state of individual actors. Executing an actor on the same NUMA node where its state is allocated minimizes memory access times. Hence, keeping actors on or close to their initial NUMA node can be beneficial.

CL and EL may conflict. An example are two data-intensive actors that are located on different NUMA-nodes and frequently exchange messages. The scheduler could optimize the CL by running both actors on the same PU. However, this would degrade EL of one of the actors because it has to access its data remotely. On the other hand, keeping each actor close to its state would result in a poor communication locality. An optimal strategy would have to analyze the trade-off between the respective memory access characteristics and the communication overhead. CAF cannot solve this challenge without support of the application developer, as the runtime environment does not have knowledge about the context of messages. For example, a message might only contain a pointer and look small, but reference a large data structure to be processed by the recipient. Consequently, we need to consider different metrics.

CAF can adjusted CL and EL by scheduling decisions at two opportunities: (1) a worker finished its job and is looking for new work, (2) an idle actor receives a message.

Work-stealing is a decentralized scheduling algorithm where each worker has its own job queue. Once the queue of a worker is drained, it picks another worker and tries to steal a work item from it. The random work-stealing [4] scheduler in CAF handles these situations as follows. When looking for new work,a victim is chosen at random among all other workers. Although very simple, this strategie ignores data locality and misses opportunities to improve the application performance.

An actor without a message in its mailbox is considered idle and its absent from all job queues. On message receipt, such an actor is scheduled at the worker of the sender. This maximizes CL at the cost of EL. As a result, an actor which relies on a large data set may be at a significant disadvantage when moved away.

2.3 Related Work

Work-stealing is a widely used for scheduling actors or tasks for example by OpenMP [16], Erlang [3], Akka [22], the Pony Language [7] and CAF.

Random work stealing (RWS) [4] scales well by following a distributed approach, it is *stable* [20] because it requires little overhead if the system is under high load, and the required information is limited to the number of victims. RWS was evaluated as a load balancing strategy between clusters connected over a wide area network (WAN) [23]. Although it performs well within a cluster, stealing work from a remote machine over a WAN link is problematic as the network introduces significant latencies. Additionally, stealing from remote cluster members is much more likely due to the (uniform) randomness when choosing a victim.

We experience a similar problem within a NUMA machine. Here, work is unnecessarily stolen from other NUMA-nodes which results in poor execution locality (EL). Previous work provides multiple improvements to compensate for high network delays and to reduce the bandwidth consumption [23]. However, none of these solutions are feasible for a NUMA-aware scheduler because they hide high network delays by prefetching mechanisms but do not consider the problem of data locality.

Scheduling algorithm such as work-sharing or random work-pushing [20] have scalability problems and no advantage for CAF. A work-sharing scheduler has a centralized job queue. Work items are enqueued at the tail and workers dequeue them from the head and execute them. This can cause contention due to the synchronization requirements. In contrast, random work-pushing is a distributed approach similar to RWS. Each worker has its own job queue. Once the amount of jobs in a queue exceeds a threshold, its worker pushes surplus jobs to another random worker as a proactive procedure. This algorithm balances the size of all queues and thus improves fairness in preemptive approaches. However, it is *unstable* since high load on all PUs leads to increasing and unsuccessful push attempts that degrade performance.

There are several approaches to work-stealing that consider data locality. Acar et al. [1] analyzed cache misses and proposed a locality-guided work-stealing algorithm for threads. They explain, that a thread should preferably be executed by a single PU to reduce the number of cache misses. This can be achieved by assigning thread an affinity for a specific PU and equipping workers with a priority-aware queue. Threads are scheduled twice, once with a normal priority at the current worker and once with a high priority at the affinity worker. Hence, a thread can be in two different job queues and it must be ensured that it is only executed once. This approach can be adapted to actors by giving actors an affinity for a worker. A drawback is the synchronization between workers to avoid repeated execution of the same job, which is a costlier for actors than for threads due to the much higher quantity of actors.

In hierarchical scheduling, work-stealing is composed with work-sharing to exploit shared caches and improve the data locality in NUMA-systems [16]. PUs that share a L2 or L3-cache are grouped together and use work-sharing to balance their workload. Once the shared queue is drained, workers try to steal items from other groups. Each steal attempt tries to acquire one item for each member of the group in order to minimize communication. This approach increases the data locality by reducing the number of remote steals and efficiently utilizes shared caches.

Class-based scheduling categorizes task based on their memory footprint [24]. In this approach, workers are equipped with a dedicated and a shared job queue. Tasks with a high memory footprint are added to the former queue and cannot be stolen while tasks that can be stolen at a low cost are added to the latter. The queue for a newly created is chosen based on factors like data size and the expected execution time. This algorithm is unsuited for CAF because memory footprints of actors and messages are opaque to the runtime system.

Quintin et al. [18] propose a probabilistic approach to increase the data locality of RWS, called Probabilistic Work Stealing (PWS). It works similar to RWS but the probability to become a victim is proportional to the inverse of the distance to the thief. This increases the data locality because the chance to become a victim increases with proximity. Although PWS was designed with a computer network in mind, the concept can be applied to a NUMA-system. A static description of the memory architecture would be enough to calculate all required information during startup. This is a desirable property as it minimizes the runtime overhead for choosing a victim. Furthermore, involvement from an application developer is not required.

The actor communication patter of *hubs* and *hub affinity groups* was introduced by Francesquini et al. [9] in the context of Erlang applications. To avoid any confusion with the Erlang terminology, we use the term actor when we refer to an Erlang process and we use the term worker for an Erlang scheduler. All proposed improvements related to this communication pattern focus on the communication locality (CL) because Erlang actors are migrated and always have the optimal execution locality (EL). A hub actor communicates with many different actors while actors in a hub affinity group mostly communicate with a specific hub. Placing a hub and its affinity group in close proximity improves the CL of the system.

To prevent the Erlang load balancer from distributing hubs and their affinity groups across distant PUs and thus decreasing the CL, the scheduling algorithm is divided into phases: *Initial Actor Placement* and *Hierarchical Load-Balancing and Work-Stealing*. In the first phase newly spawned actors are grouped and placed at a specific worker. On spawning, the application programmer gives the Erlang virtual machine (VM) a hint whether the actor is a hub or a regular actor. While a hub receives its own affinity group, a regular actor inherits the affinity group of its parent. The VM spreads hubs over the available workers, e.g., in a round robin fashion, and places regular actors close to their hub. If an actor is executed for the first time, it stores the current NUMA-node as its home-node to provide the scheduler with its preferred location in the future.

Using these information, the periodic load-balancer tries to migrate actors back to their home-node at first. It increases the migration radius if this is not sufficient to balance the system, first within and then across NUMA-nodes. Work-stealing in Erlang works similar to PWS [18] algorithm. The algorithm takes the memory architecture into account by preferring direct neighbors as a victim over distant ones.

Although the Erlang VM differs in many ways from CAF, the described concepts can be adapted. In contrast to Erlang, CAF actors are not migrated between workers. Hence, storing the home-node and considering it on scheduling can lead to a big performance boost even without the concept of hubs. A periodic load balancer could improve the performance of CAF based applications by reducing the number of steals. However, it is much more important for Erlang which implements preemptive scheduling and balancing work queue sizes is crucial for fair allocation of hardware resources.

3 Locality-guided Scheduling

We now present our locality-guided scheduling strategy for multicore systems with heterogeneous memory architecture. The strategy consists of two mechanisms, a weighted work-stealing approach that preferably picks victims from memory vicinity, and a soft actor pinning that schedules actors close to their initial worker for facilitating fast access of state.

Random work stealing favors full resources utilization at the cost of locality when moving actors between workers in the system. The weighted work stealing is likely to preserve execution locality when stealing. Actor pinning prevents actors to move away from their data during rescheduling. While pinning can be deployed without weighted work stealing, the reverse does not hold. Weighted stealing correlates actors with their probable queue location that is not given without pinning.

3.1 Weighted Work Stealing

Fully randomized work stealing leads to poor execution locality because it ignores memory access costs. We adjust the probabilities for picking a victim based on the NUMA architecture in the same way Probabilistic Work Stealing (PWS) adjusts probabilities based on the network architecture in a cluster [18]. The probability for picking victims is proportional to the inverse of the distance to the thief. The distance can be defined by the number of hops between the thief node and the victim node.

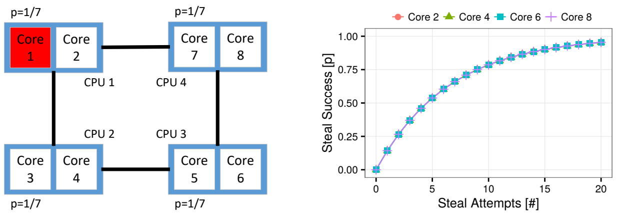
We contribute a practical approach to weighted work stealing with minimal runtime overhead. On program start, hardware information are gathered and each worker (thief) sorts all other workers (potential victims) according to their distance into the groups $g_0 \subseteq \ldots \subseteq g_k$, where the index correlates to the maximum distance. The group g_0 only contains direct neighbors. Note that the definition of neighborhood varies on different platforms and can depend on shared cache levels or shared memory banks. The group g_1 contains all direct neighbors as well as all workers with distance 1, and so on. Finally, g_k contains all potential victims. Stealing from groups with lower index correlates to faster execution times, since stealing from distant workers causes expensive memory exchange.

Once a worker runs out of work, it becomes a thief and tries to steal work items from all victim groups in increasing order. The steal attempts per group depend on the size of the group. The thief performs the lowest number of steal attempts on g_0 but picks victims from the final group g_k indefinitely. A worker does not remember the group where it last picked its victim from and always starts anew at g_0 after a successful heist.

On platforms with uniform memory access, a single group is created that includes all workers. The same approach is taken in case reading the NUMA-node layout fails at runtime. In both cases, our scheduling is equivalent to the classical random work stealing.

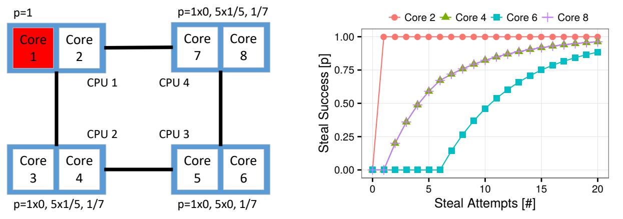
When a thief has picked a victim with a non-empty queue it steals the tail element, i.e., the item with the longest wait duration. Stealing multiple work items can be beneficial for homogeneous item runtimes to reduce future stealing. However, CAF has no a priori knowledge about the cost to process a work item and thus cannot estimate whether stealing more than one item at a time is beneficial. In the worst case, a thief steals expensive work items that the victim than steals back later. Looking for a work item with specific properties, e.g., one with a nearby home processing unit [9], would require an expensive search in the job queue and cause additional synchronization overhead. Although a specialized data structure can reduce the search cost for rare stealing events, it would be less optimal for the general program execution. Additionally, workers enqueue newly spawned actors at the head of their queue to benefit from caching mechanism. As a result, work items at the tail of a queue are less likely cached and should therefore be stolen with higher preference.

As an example, consider a system with CPUs, each equipped with two cores, and connected in a ring. Figure 1(a) shows this layout annotated with the probability for each core to successfully steal a work-item from core 1 (marked red) when using random work stealing (RWS). Cores of a CPU are direct neighbors and the stealing distance is defined by the number of CPU hops to reach the victim. In this case, the worker of core 1 is a hot spot with many enqueued work items. Other workers are idle and try to steal these items. The probability for a successful steal is one out of seven because the system has seven other cores a worker can steal from. Figure 1(b) plots the probability for a successful steal as a function of the number of consecutive attempts. Since all cores have the same probability, the graphs overlap. When considering locality-aware stealing these probabilities change. Figure 2 depicts the setup with adjusted probabilities. Here, the probability for a successful steal depends on the distance between thief and victim and the number of stealing attempts before the thief increases its radius. From the perspective of *core 6*, all potential victims are divided into three groups $g_0 = \{5\}$, $g_1 = \{3, 4, 5, 7, 8\}$, $g_2 = \{1, 2, 3, 4, 5, 7, 8\}$. On the first attempt, the success rate to steal a work item is 0 because the only worker in g_0 does not have any jobs either. The next five steals attempts have the same expectation with group g_1. Finally, after 6 unsuccessful attempts, the success rate increases to one out of seven. These probabilities depend on the location of the core and are plotted in Figure 2(b). Here, core 2 will immediately steal work, while cores 4 and 8 will have to increase their radius once, and core 6 will have to increase its radius twice. This matches the targeted behavior of our locality-aware approach as it gives direct neighbors a higher probability to steal work items than distant ones.



(a) Core layout with probabilities (b) Probability vs. steal attempts

Figure 1. Chances for successfully stealing a work-item from core one using *random* work stealing.



(a) Core layout with probabilities (b) Probability vs. steal attempts

Figure 2. Chances for successfully stealing a work-item from core one using *locality-aware* work stealing.

3.2 Soft Actor Pinning

Actor pinning improves the execution locality (EL) by fixing actors to workers in close proximity of their data. The pinning strategy implemented our scheduler is static soft pinning that is automatically handled by the framework.

Similar to the approach of Francesquini et al. [9], the algorithm is divided into the phases *Initial Actor Placement* and *Scheduling*. Following this idea, newly spawned actors are placed at a specific worker during the *Initial Actor Placement* phase. CAF has two options to place an actor in the worker pool: either in a round robin fashion or at the worker of its parent. The former evenly distributes actors to balance the workload while the latter schedules actors for fast execution with cache optimization in mind. In both cases, an actor stores the current worker persistently as its home processing unit (HPU) on first execution as proposed by Acar et al. [1]. Thereafter, the actor is pinned to this node (*static*).

In the *Scheduling* phase, an actor can be stolen and executed by an arbitrary worker for balancing reasons (*soft*)–diverging from the original algorithm. However, it moves

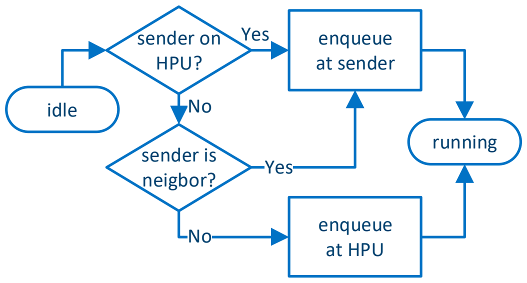


Figure 3. Flowchart to determine the worker during the scheduling phase.

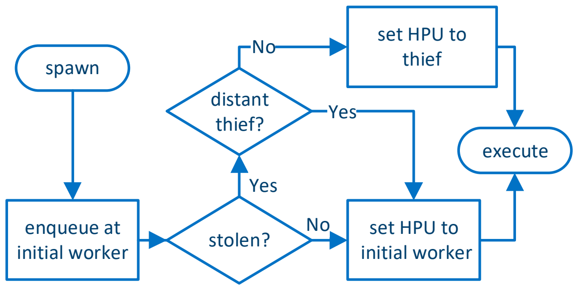


Figure 4. Flowchart to determine the HPU during actor initialization.

back to its HPU for subsequent executions. For this reason, an idle actor that receives a message is scheduled at its HPU to guarantee an excellent EL. As an exception, an idle actor will be scheduled at the PU of the sender if it is a direct neighbor to the HPU of the receiver, thus maintaining a good EL while improving the CL. Here, we trade an optimal EL for an optimal CL because the idle actor is scheduled at the PU of the sender instead of its HPU. Figure 3 shows where an idle actor is enqueued when it receives a message.

This approach requires no additional effort form an application developer (*automatic*). Moreover, it has little computational and memory overhead. The only additional information an actor has to store is its HPU. Note that an actor can allocate memory on each execution at which point it acquires memory from the NUMA-node where it is currently executed (first-touch). If an actor jumps between NUMA-nodes for balancing reasons, its memory might be scattered across different NUMA-nodes which causes a degradation in EL. A dedicated memory allocator could allow application developers to ensure that an actor accumulates all its memory from NUMA-nodes of its HPU and thus avoid this behavior.

Actor pinning largely enhances the importance of initial actor placement across workers. An uneven placement may lead to frequent stealing as actors return to their initial worker after execution. This inclination to return to a potentially unbalanced state can significantly impact the performance. To address this problem, actors do not inherit the HPU of a parent. Instead, the HPU is assigned at first execution. This allows other workers to steal newly spawned actors, thus balancing the system.

In general, actors can be pinned to a location such as a single core, a group of cores sharing a cache level or to a specific NUMA-node with a hard [24] or a soft [1] constraint. Both cases prohibit scheduling of pinned actors at another location. However, soft pinning still allows workers to execute stolen actors. In this case, the execution on a distant worker is only temporary and the actor jumps back to its home node after the execution. While both strategies are suitable for actors with heavy memory accesses and I/O interactions, we chose soft pinning for CAF because hard pinning can easily lead to performance degradation as a result of an imbalanced workload.

Alternatives to an automatic pinning strategy are semi-automatic and manual pinning. A semi-automatic approach allows programmers to provide hints to the scheduler [9] such as tightly coupled actors or dependencies on specific I/O devices. The scheduler can use this knowledge for optimization according to its strategy. Specific problems can be addressed well with this approach, e.g., pinning actors which require access to I/O devices like GPUs to the appropriate NUMA-node. However, this is impractical as a generic approach since it is not portable and hard to maintain for a larger code base. Automatic pinning [1] does not requires specialized knowledge of the programmer by transparently handling pinning decisions. A static strategy could be pinning all actors to their initial workers. This is a good general purpose approach because actors initialize their state on their first execution when the required memory is allocated from the host NUMA-node. Thereafter, this node has the best EL for this actor. A dynamic strategy could profile the relationship between actors and decide at runtime which groups of actors are closely coupled and should be executed by the same processing unit. CAF implements a static strategy to avoid the additional complexity inherent to profiling.

3.3 Discussion: Soft-pinning and Sleep Intervals

The improvements to execution locality offered by actor pinning comes with some trade-offs. While an idle actor that receives a message was previously *pulled* to the worker of the sender, it is now *pushed* to its HPU. Pulling an actor ensures that the worker that receives the work is awake and can directly react to the new job. In contrast, the push approach can enqueue work into the job queue of a sleeping worker—workers sleep shortly to reduce the system load and contention of work queues if their queue is empty and they do not find work to steal. In such a scenario, the execution of the actor is delayed until the respective worker wakes up or it is stolen.

The benchmark discussed in Section 4.2 displays a scenario where this behavior impacts performance: a system hosts two actors that exchange message in a ping-pong pattern. Both actors are placed at the same worker when spawned and immediately scheduled for execution to initializes their behavior and prepare for future messages. Due to unfavorable timing one actor might be stolen by a worker on a different NUMA-node before its first execution. As a result, the actors don't have neighboring HPUs and are never scheduled at the worker of their communication partner. Instead, they are pushed to their HPU on message receipt. While waiting for a reply, the respective worker becomes idle and goes to sleep, thus introducing a delay to each message exchange. To mitigate this effect, we restricted the initial definition of the HPU to direct neighbors of the worker where the actor is initially placed as shown in Figure 4. If an actor is stolen from a distant worker before it could set its HPU, it uses its initial worker as HPU. Otherwise the HPU is set to the thief. This ensures that tightly coupled actors are not "ripped apart" when spawned and maintain reasonable proximity instead. Note that this problem does not occur for actors pinned to the same worker or to a direct neighbor as they can be executed by the same worker in both cases.

Message delay as a result of sleeping workers is not unique to the scenario discussed here and might still appear with different initial configurations. However, a real world application is unlikely to run into such a problem for multiple reasons: (1) workers only sleep if the actor system has a low workload, (2) a sleep time of 50 μs is the maximal execution delay for an actor which can still be stolen in the meantime (although longer sleeps may occur if the actor system has a low workload over a long period of time), and (3) the sleep interval can be reduced or deactivated.

4 Evaluation

Our first benchmark *Matrix Search* focuses on a data-intensive task to showcase the benefits of locality-guided scheduling over our previous scheduling approach which uses RWS and focuses on CL. A fixed number of actors s each solve word-finding puzzles distributed by a smaller number of coordinators c.

Solving a puzzle requires an actor to find a sequence of characters assigned by a controller in a local matrix of random characters. For this purpose, only matches along columns are valid. Since the matrices are written row-wise into memory, this bypasses the prefetching mechanism of the CPU and increases the complexity of the data access. Additionally, an uneven distribution of puzzle complexity along with a large number of actors ensures irregular rescheduling of actors. A controller only distributes a new challenge after a previous one was solved. As a result, the controller is idle while its actors are busy and it has to be rescheduled whenever a new challenge is requested. Note that a controller and the actors it manages are not forced to have the same home processing unit (HPU), although this may happen by chance.

The performance of this benchmark greatly relies on the access characteristics of searching actors to their matrices. By soft pinning actors and taking locality into account when stealing actors we vastly increase the chance for a good execution locality between an actor and its matrix and thus improve the performance. While the benchmark itself is an artificial scenario, it showcases the effect that consideration of locality has on runtime behavior.

Figure 5 depicts the runtime of the matrix search benchmark in seconds as a function of the workers of the scheduler. It compares the locality-guided scheduling (LGS) to the CL focused scheduling approach (CLS) that does not take NUMA-related optimizations into account, both implemented in CAF. All measurements were performed on a server with four *AMD Opteron 6376* processors, clocked at 2.3 GHz. Each processor is divided in two NUMA-nodes, each consisting of 8 cores and 64 GB of main memory. This adds up to a total of 64 PUs spread over 8 NUMA-nodes. The server is powered by a Linux distribution (kernel version 3.16.7) with default NUMA-settings (first-touch). Each measurement is repeated 10 times for a statistical significance, plotting the mean as well as error bars that show the 95% confidence interval.

For the measurements, we use $c = 15$ controllers, where each controller spawns 15 actors, which adds up to $s = 225$ matrix-searching actors. Actors are assigned to workers in a round robin fashion. To change the number of workers, we enable cores on the host in steps of four and provide the CAF scheduler with an equal amount of worker threads. For a configuration of four cores, each activated core is hosted on a different NUMA-node. These NUMA-nodes are filled up until each node has eight active cores before cores on the remaining NUMA-nodes are activated.

Both graphs are nearly overlapping for configurations of up to 12 workers. The amount of actors scheduled across these workers is enough to keep them busy while mostly executing actors local to their NUMA node. Thereafter, the LGS outperforms the CLS with a performance gain of up to 26.6%. The difference in runtime increases up until 28 workers are active and stays in a similar range thereafter with a few variations. Our guess is that the capacity of the bus systems is maxed at this point.

Overall, LGS outperforms CLS in this benchmark due to its consideration of EL. The more workers are active, the greater the chance that actors are executed on a different node. This is a result of the increased chance that work is stolen as the amount of initial actors per worker decreases and the uneven processing time has a greater impact. Since actors return to their HPU under LGS and are usually stolen from a closer worker, the locality-aware approach exhibits better performance.

Matrix Search heavily favors data access and reveals what can be achieved in general. The subsequent benchmarks are

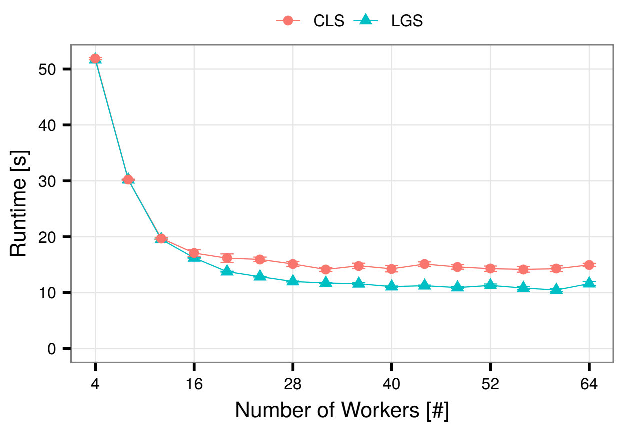


Figure 5. Actors solve word finding puzzles to simulate a data-intensive task (Matrix Search).

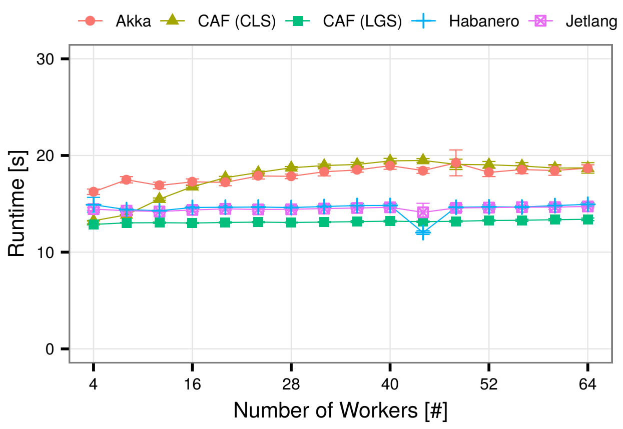


Figure 6. Actors synchronize access into a linked list through a central coordinator (Savina: Concsll).

translated from the Savina benchmark suite [13] and are not necessarily written with data locality in mind. They showcase how our LGS strategy performs in more generalized scenarios and how it compares to our CLS approach as well as to other implementations of the actor model. The same test environment is used for all benchmarks.

We translated 23 of the 30 Savina benchmarks to C++[1]. These benchmarks cover a wide range of concurrency patterns and support multiple Java based actor libraries including Akka [22], Habanero-Java library [12] and Jetlang [19], which is the set we use for comparison with CAF. Not all benchmarks could be translated as some rely on Java-specific libraries. The benchmarks are divided in the classes micro-benchmarks, classical concurrency problems and parallelism benchmarks. Problems range from *Ping-Pong* and *Dining Philosophers* to *Quicksort* and the *N-Queens Problem*. The benchmark suite was introduced by Imam et al. [13] with the goal to provide a general set of benchmarks to compare different implementations of the actor model. All benchmarks allow configuration of the problem size to tune the computation time as well as the degree of concurrency, if possible.

The remainder of this section discusses three examples picked from the Savina benchmarks and shortly summarize our observations on the performances of LGS for the whole suite.

4.1 The Delicate Difference Between Data and Communication Intensive Applications

In our next measurement we compare a data-intensive application to a communication-intensive application. For this purpose, we use the benchmarks *Concurrent Dictionary* (Condict) and *Concurrent Sorted Linked-List* (Concsll) from the Savina suite. Both provide a central data structure encapsulated in an actor. A number of other actors accesses the data structure by sending read and write requests. Condict uses a dictionary with a read/write complexity of $O(1)$ while Concsll uses a sorted linked list with complexity $O(N)$.

Figure 6 depicts the makespan in seconds as a function of the number of scheduler workers for the Concsll benchmark. We use the default configuration parameters except for the number of entities set to $e = 20$ and the number of messages set to $m = 8000$. Most frameworks exhibit similar performance except for Akka and CAF (CLS) which perform worse than the rest. The best performance is shown by CAF (LGS), which outperform the other CAF scheduler by up to 32.5%. Although this performance gain is even higher than in our showcase Matrix Search, it is not solely caused by optimizing the data locality but rather a result of a strong workload imbalance that does not occur for LGS. Figure 7 shows results of the Condict benchmark for $e = 100$ and $m = 50000$. Here, both CAF scheduling strategies admit good performance while CLS is 17.6% faster on average. The remaining frameworks perform poorly, where most show a strong runtime increase at the beginning and slightly rise thereafter.

Despite the similarities between Concsll and Condict, LGS only shows better performance in the former benchmark. This is due to the different characteristics of the data structure and the resulting memory access. The data actor of Concsll is more sensitive to EL than Condict because it traverses the sorted linked list on every read and write access. In contrast, the dictionary has a constant access time and a low access complexity. As a result, solely optimizing for CL pays off for the Condict benchmark, as done by CLS, instead of finding a trade-off between CL and EL, as done by LGS.

[1]https://github.com/shamsimam/savina

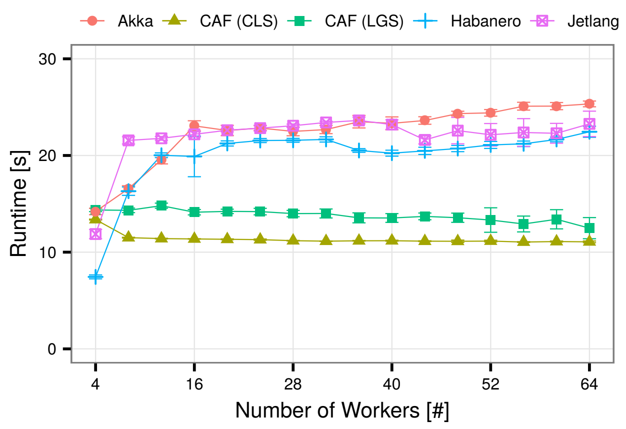


Figure 7. Actors synchronize access into a dictonary through a central coordinator (Savina: Condict).

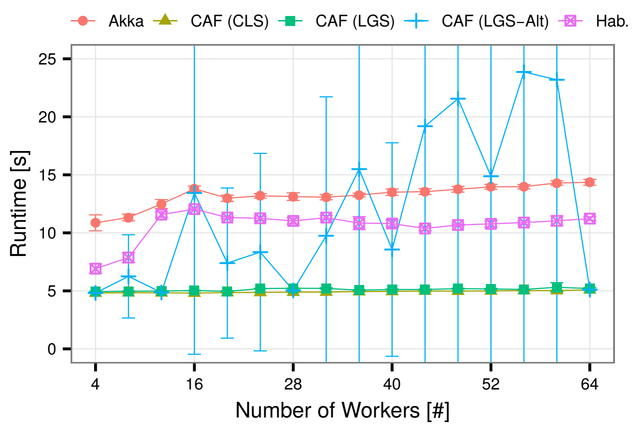


Figure 8. Two actors repeatedly exchange messages in a ping-pong style communication pattern (Savina: Ping Pong).

4.2 Pushing Actors to Sleeping Workers

The last benchmark we present is the Savina Ping-Pong benchmark, which heavily favors communication locality. As the name suggest, it deploys two actors that exchange a defined number of messages, in our case $2 * 10^6$.

Figure 8 shows the runtime as a function of the number of workers for the benchmark. Both CAF deployments, CLS and LGS, outperform the other frameworks under test. The graph plots an additional measurement for CAF: LGS-Alt. LGS-Alt exhibits enormous error bars and a very unstable runtime behavior. This is an artifact of undesirable scheduling that can happen when jobs are pushed to a sleeping worker. This behavior was discussed in Section 3.3 alongside a mitigation strategy that is implemented for the LGS measurements.

Part of the mitigation strategy prevents actors from adopting a distant node as a HPU if they are stolen before their first execution. As a result, the LGS avoids intermediate sleeps from involved workers as well as communication between distant NUMA-nodes in such situations. To show that behavior of LGS-Alt is not only a result of the communication distance we performed a measurement where we prevented workers from sleeping. Under those condition, LGS-Alt still performed about 30% worse than LGS, but exhibited a stable runtime behavior.

The Ping Pong benchmark stressed a performance problem that could occur under specific situations. After implementing a mitigation strategy, LGS only shows slightly worse performance than CLS although the benchmark mainly relies on communication locality. Incidentally, the adjustments to LGS also ensure an optimal execution locality for this benchmark.

4.3 Summarizing Benchmarks of the Savina Suite

To summarize the overall performance of LGS compared to CLS in the Savina benchmarks, we performed measurements for both schedulers using 64 workers. The baseline (100%) signifies the mean runtime of CLS over 10 measurements. A lower percentage shows a better performance in favor of LGS, e.g., LGS finished after 63% of the time required by CLS for the *Apsp* benchmark. Note that the graph only shows an overall trend and does neither provide information about error distribution nor scalability.

The benchmarks *Apsp*, *Bitonicsort* and *Concsll* show excellent results. They benefit not only from an increased EL but also from a better load balancing between workers. Whether a benchmark is balanced depends on the interplay between the workload and the scheduling algorithm. *Bndbuffer* is an example where LGS leads to scalability problems.

The *Big* benchmark only profits from CL while EL is nearly irrelevant. It represents an *N* to *N* ping pong scenario, a best-case candidate for CLS. Keeping EL in mind prevents LGS from pulling actors to their communication partners, a strategy that greatly benefits CLS in this measurement.

Overall, theses benchmarks confirm that LGS can lead to significant performance gains for suitable problems and we reached the expected gains above 20%. However, there are scenarios where trading between EL and CL significantly degrades performance. This is clearly visible in the last seven benchmarks towards the right side which exhibit a runtime increase of more than 10% and even up to a few 100%.

5 Conclusion and Outlook

Cores on modern processor architectures do not have uniform access to memory. Instead, cores are bundled with caches and memory banks on different NUMA-nodes, thereby experiencing performance that depends on data proximity. The architecture is accessible to developers via a NUMA API so that tasks can be kept in close proximity to their active memory. This can significantly improve performance.

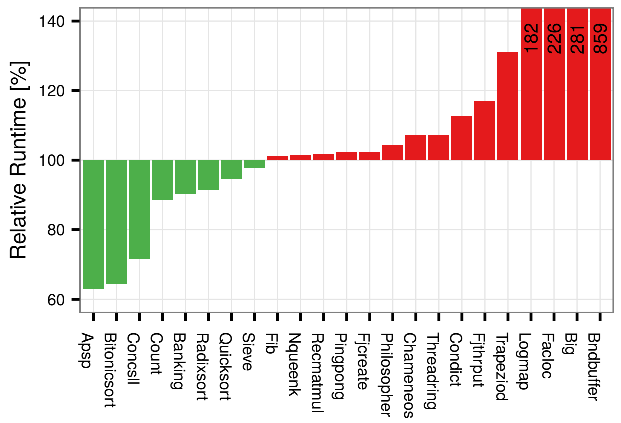


Figure 9. Comparison between LGS and CLS in the C++ Savina benchmarks with 64 workers.

In this work, we introduced locality-guided scheduling (LGS) which exploits knowledge about the host architecture to improve scheduling for actor-based applications in CAF. LGS shifts the strategy of our scheduler towards data locality and aims at a trade-off between communication and execution locality. For this purpose, it combines a weighted work stealing approach with actor pinning.

Extensive benchmarks confirmed that LGS increases the performance of data-intensive task as expected. We measured a performance gains of up to 25%. Using the Savina benchmark suite, we explored the LGS behavior in a wide range of diverse scenarios. Here, LGS improved the performance of 8 out of 23 benchmarks over a scheduler focusing on pure communication locality. However, the focus on execution locality had significant impact on the performance in a range of benchmarks. While this is partially explained by missing optimization of communication locality in favor of execution locality, it suggests that LGS is suitable as an optional scheduling strategy for CAF.

Our future work will proceed in two directions. First, we will improve LGS and analyze performance further. Incorporating scheduling hints from developers could offer more fine-grained control and help the scheduler to avoid negative impact of EL optimizations. Second, tuning the trade-off between communication and execution locality in LGS requires a more detailed analysis. Comparison with other NUMA-aware frameworks could provide further insight into the performance and behavior of LGS.

Acknowledgments

This work was supported in parts by the German Federal Ministry of Education and Research within the projects Scalecast and X-Check.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The Data Locality of Work Stealing. In *Proc. of the 12th An. ACM Symposium on Parallel Algorithms and Arch. (SPAA '00)*. ACM, NY, USA, 1–12.

[2] David Applegate and William Cook. 1991. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journ. on comp.* 3, 2, 149–156.

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

[4] R. D. Blumofe and Ch. E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept.), 720–748.

[5] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2016. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures* 45 (April 2016), 105–131.

[6] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. 2013. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proc. of the 4rd SPLASH '13, WS AGERE!* ACM, NY, USA, 87–96.

[7] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proc. of the 6th SPLASH '15, WS AGERE!*. ACM, NY, USA, 1–12.

[8] Peter J. Denning. 2005. The Locality Principle. *Commun. ACM* 48, 7 (2005), 19–24.

[9] Emilio Francesquini, Alfredo Goldman, and Jean-François Méhaut. 2013. Actor Scheduling for Multicore Hierarchical Memory Platforms. In *Proc. of the 12th ACM SIGPLAN Workshop on Erlang (Erlang '13)*. ACM, New York, NY, USA, 51–62.

[10] Fabien Gaud, Baptiste Lepers, Justin Funston, et al., 2015. Challenges of Memory Management on Modern NUMA Systems. *Commun. ACM* 58, 12 (2015), 59–66.

[11] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. of the 3rd IJCAI.* Morgan Kaufmann, San Francisco, CA, USA, 235–245.

[12] Shams Imam and Vivek Sarkar. 2014. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *PPPJ.* ACM, 75–86.

[13] Shams Imam and Vivek Sarkar. 2014. Savina – An Actor Benchmark Suite. In *Proc. of the 5th SPLASH '14, WS AGERE!* ACM, NY, USA, 67–80.

[14] Shams M. Imam and Vivek Sarkar. 2012. Integrating Task Parallelism with Actors. *SIGPLAN Not.* 47, 10 (Oct. 2012), 753–772.

[15] Kirk L. Johnson. 1992. The Impact of Communication Locality on Large-scale Multiprocessor Performance. *SIGARCH Comput. Archit. News* 20, 2 (1992), 392–402.

[16] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. 2012. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *Int. J. High Perform. Comp. Appl.* 26, 2, 110–124.

[17] M. Pericas, A. Cristal, R. Gonzalez, D. A. Jimenez, and M. Valero. 2006. A decoupled KILO-instruction processor. In *The 12th Intern. Symp. on High-Perform. Comp. Arch.,'06.* Springer, Berlin, Heidelberg, 53–64.

[18] Jean-Noël Quintin and Frédéric Wagner. 2010. Hierarchical Work-stealing. In *Proc. of the 16th Intern. Euro-Par Conf. on Parallel Processing: Part I (EuroPar'10).* Springer-Verlag, Berlin, Heidelberg, 217–229.

[19] Mike Rettig. 2012. Jetlang. code.google.com/p/jetlang. (April 2012).

[20] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. 1992. Load Distributing for Locally Distr. Systems. *Computer* 25, 12, 33–44.

[21] H. Topcuoglu, S. Hariri, and Min-You Wu. 1999. Task Scheduling Algorithms for Heterogeneous Processors. In *Het. Comp. WS. (HCW '99) Proceedings. 8th.* IEEE Comp. Soc., DC, USA, 3–14.

[22] Typesafe Inc. 2017. Akka Framework. http://akka.io. (August 2017).

[23] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. 2001. Efficient Load Balancing for Wide-area Divide-and-conquer Applications. *SIGPLAN Not.* 36, 7 (2001), 34–43.

[24] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu. 2014. Optimizing Load Balancing and Data-Locality with Data-aware Scheduling. In *2014 IEEE Int. Conf. on Big Data.* IEEE, DC, USA, 119–128.