Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Master Thesis

Sebastian Wölke

Optimizing Data Locality in CAF

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sebastian Wölke

Optimizing Data Locality in CAF

Master Thesis eingereicht im Rahmen des Masterprüfung

im Studiengang Master Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Professor: Prof. Dr. Thomas C. Schmidt
Prof. Dr. Wolfgang Fohl

Eingereicht am: January 14, 2018

Sebastian Wölke

Thema der Arbeit
Optimizing Data Locality in CAF

Stichworte
Aktormodell, Scheduling, Datenlokalität

Kurzzusammenfassung
Die Speicherarchitektur moderner Mehrprozessorsysteme ist hierarchisch strukturiert. Mehrere Ebenen von Caches und eine *Non-Uniform Memory Access* (NUMA) Architektur wurden eingeführt um die Zugriffszeiten von den CPU Cores auf die benötigten Daten zu verringern. Dies führt zu inhomogen Ausführungsgeschwindigkeiten von Anwendungen mit einer Abhängigkeit vom Ort der Daten. Durch die Berücksichtigung dieses Lokalitätsprinzips kann die Geschwindigkeit von Anwendungen und Software-Bibliotheken, wie zum Beispiel bei dem C++ Actor Framework (CAF), deutlich verbessert werden. CAF ist eine Implementierung des Aktormodells, ein mächtiges Entwurfsmuster für verteilte und nebenläufige Anwendungen. In CAF wird *Random Work-Stealing* (RWS) als Standard-Scheduler genutzt. RWS skaliert exzellent, ist einfach gehalten und benötigt nur sehr wenig Informationen über das System. Dies hat den Nachteil, dass es keine Wissen über die Speicherarchitektur hat, das Lokalitätsprinzip ignoriert und Möglichkeiten versäumt die Performanz von Anwendungen zu verbessern.

In dieser Arbeit entwickeln wir einen Scheduler, der das Wissen über die Speicherarchitektur ausnutzt um die Performanz von aktorbasierten Anwendungen zu verbessern. Wir implementieren und analysieren den Scheduler in CAF unter Berücksichtigung eines Kompromisses zwischen der Kommunikations- und der Ausführungslokalität. Die Kommunikationlokalität beschreibt die Distanz zwischen kommunizierenden Aktoren während die Ausführungslokalität die Distanz zwischen dem auszuführenden Aktor und seinen abgelegten Daten beschreibt. Ausführliche Analysen zeigen, dass datenintensive Anwendungen bis zu 44% schneller auf einer 64 Core NUMA-Maschine laufen können.

Sebastian Wölke

Title of the paper
Optimizing Data Locality in CAF

Keywords
Actor Model, Scheduling, Data Locality, NUMA

Abstract
Memory architectures of modern processors are hierarchically structured. Multiple level of caches and a non-uniform memory access architecture (NUMA) are introduced to reduce access latency from processing units to their current working set. These lead to inhomogeneous performance characteristics depending on where the data is located. Optimizing the data locality of applications or libraries like the C++ Actor Framework (CAF) can significantly improve the performance. CAF is an implementation of the actor model. It is a powerful software pattern for concurrent and distributed computing. CAF is designed for using multiple, exchangeable schedulers with a default choice of random work-stealing (RWS). RWS is excellently scalable, and by choosing a random victim scheduling is kept simple with minimal information required. On the downside, it is unaware of the memory architecture, ignores data locality and misses opportunities to improve the application performance.

In this thesis, we contribute a locality-guided scheduling that exploits knowledge about the host system to improve 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 44% on a 64 core NUMA machine.

Contents

1 Introduction

Concurrent programming becomes continuously more important as the number of cores per CPU 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 [1]. 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) [2, 3] 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 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. Taking data locality into account can improve the performance of applications that utilize heterogeneous memory architectures. We focus this work on optimizing the scheduler in CAF which promises to have a significant impact on the data locality.

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.

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. LGS considers *communication locality* (CL) [5], the locality of communicating actors, and *execution*

locality (EL) [6], the locality between a worker and the data of the actor it executes. Locality describes the arrangement of entities and data over CPUs, caches, and memory banks. We combine *weighted work-stealing* and *actor pinning* that enables LGS to find a trade-off between the two localities. The former enables workers to prefer victims with close memory proximity while the latter improves EL by scheduling actors near their state.

We evaluate LGS in two steps. First, we benchmark its performance in a data-intensive task to quantify the case in suitable scenarios. Thereafter we examine the performance of LGS in the actor benchmark suite Savina [7] which implements a wide variety of concurrency patterns for actor systems.

1.1 Thesis Outline

The thesis is organized as follows. Chapter 2 introduces the actor model and the C++ Actor Framework. Subsequently, it describes the fundamentals of data locality and scheduling. Chapter 3 discusses scheduling challenges and design constrains along with related work. Chapter 4 describes locality-guided scheduling in detail before discussing the implementation in Chapter 5. The scheduling strategy is evaluated in Chapter 6. Finally, Chapter 7 concludes and gives an outlook to future work.

2 Background

In this chapter, we introduce the actor model and an implementation of it called the C++ Actor Framework (CAF) which we use to test and evaluate our scheduling algorithm. Subsequently, we give an overview on non-uniform memory architectures and discuss why taking data locality into account can improve the performance. Next, we introduce scheduling in CAF and survey side effects of locality related optimizations.

2.1 The Actor Model

The Actor Model is a software pattern for concurrent and distributed programming. The model introduces independent and isolated entities called actors which interact via message passing. A classic actor [8] typically responds to a received message by (1) spawning new actors, (2) sending messages to other actors or (3) changing its own behavior. An actor is a composition of an inbox, a behavior and a state which is processed by an execution unit. Messages which are used for exchanging data, sending commands and requesting results of a calculation upon reception are stored in the inbox of the actor. The inbox maintains the order in which messages are received, but does not necessarily define the order in which they are processed. An actor sequentially selects a single message from its inbox which suits its current behavior and processes it. Unsuitable messages may be dropped, skipped for later usage or generate an error. While processing a message, an actor may change its state and adjust its behavior for future messages. An execution unit may be an exclusive thread or an arbitrary and temporary assigned thread.

The actor model has clear benefits compared to low-level multithreading programming. Using independent and fully isolated actors eliminates the possibility of low-level race conditions, because an actor can only change its own state. Moreover, low-level deadlocks are impossible, because accesses to resources do not have to be locked from other actors. An actor is a lightweight entity which represents a logical task and is independent from the creation and destruction of heavy threads. Hence, it is feasible to start actors even for small tasks which allows fine-grained parallelization when scaling programs from one to many cores. Load balancing is done automatically and thus relieves the programmer of this task.

Error handling in concurrent and distributed systems is a complex challenge. The actor model allows actors to monitor one another and link their fate [9]. When a monitored actor dies, all monitoring actors are notified with a message. Either, the monitoring actor is able to handle the error, e.g., by restarting the faulty actor, or it dies as well to escalate the error. This concept helps to design resilient and fault tolerant software, because problems are propagated throughout the whole system or subsystem and it allows to re-establish faulty parts in a well defined way.

The actor model was introduced by Hewitt et al. [1] in the context of artificial intelligence and was refined later by Gul Agha [10]. While the first implementation was PLASMA [11], today many actor libraries exits. For example, for Java and Scala the libraries Akka [12], Habanero-Java [13], Jetlang [14] and Scalaz [15] exist among others. For C++11 the libraries Theron [16], Charm++ [17] and CAF [3] are provided. Furthermore, languages like Pony [18] and Erlang [19] use the actor model as a fundamental primitive.

2.2 The C++ Actor Framework

The C++ Actor Framework (CAF) [20, 21, 22, 3, 23, 24] is an open source implementation[1] of the actor model written in C++11. It combines a high level abstraction with a native program execution. CAF provides lightweight, sub-thread actors with a low memory footprint and an efficient message passing layer. This allows to scale applications from small IOT devices [23] up to computing at cluster level.

CAF implements the basic actor primitives `spawn()`, `send()` and `become()`. It extends them with convenient functions, features and building blocks like transparent network abstraction, group communication, management for groups of workers, concurrency-save standard output, composable actors and streaming support. Actors can be declared as a function or a class and are constructed with the method `spawn()`. CAF supports both statically and dynamically typed actors with asynchronous or blocking message handlers. Statically typed actors have a predefined message interface which is announced to the C++ type system. This allows to check the compatibility between senders and receivers at compile time. In contrast, dynamically typed actors have low programming overhead and allow rapid prototyping.

The CAF runtime manages a preconfigured number of worker threads which executes the actors, while a user-space scheduler assigns the actors to the workers. An actor in execution dequeues a message from its inbox that matches its current message interface. Next, it processes the message and then returns the control back to the scheduler cooperatively. While processing

[1] https://github.com/actor-framework

a message, an actor can `send()` messages to other actors in an asynchronous fashion or send a message with the method `request()` and wait for the response message before processing other messages. An actor can also change its behavior for future messages with the method `become()` which is a convenient way to implement a state machine.

2.3 Non-Uniform Memory Access Architectures

The processing speed of a computer system highly depends on how fast CPUs can retrieve data for their calculations. In this thesis, we analyze and try to optimize this data access on software level. For such optimizations we need to understand the characteristics and peculiarities of a computer system and require insights into the physical memory architecture. For simplicity and unification, we follow the example of hwloc [25] and use the term *processing unit* (PU) for the smallest processing element on a computer. A PU could signify a processor with a single core or a hardware thread in a superscalar processor.

The memory architecture is a system design to store and deliver data. In common multicore systems memory is hierarchically structured. The primary storage also called main memory is shared by all CPUs and accessible via a unified address space. This design is called a *shared memory architecture* [26] and can be further divided into designs like the *distributed memory architecture* [27] where each CPU has its own private main memory. Shared memory architectures can be classified in systems with *uniform* and *non-uniform memory accesses* (UMA, NUMA) [28]. A UMA-system guarantees same access times to each memory region from each PU. In contrast, NUMA-systems do not have such a guarantee. On the one hand, this complicates the design of performance critical software. On the other hand, it allows to bundle PUs together with memory banks into NUMA-nodes which in turn allows commodity hardware to scale linearly with the number of available PUs as long as the executed software threads work on their local memory regions. NUMA-nodes are connected via data buses (links) such that each PU can transparently access the memory of other nodes—although with varying access times. These links form an interconnection network which exists in various topologies. If NUMA-nodes are not connected in a full mesh, then multiple hops are necessary to reach all memory regions. The more hops are required to access another NUMA-node, the slower the memory access becomes. Load on links along the path can further slow down access times. Figure 2.1 shows a *Twisted Ladder Topology* [29] connecting 8 NUMA-nodes, each consist of local main memory and 8 PUs. The number of hops is limited by this topology to a maximum of three.

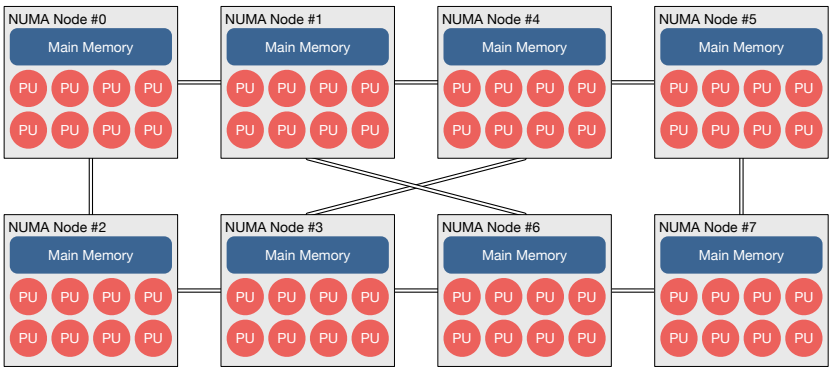


Figure 2.1: The NUMA architecture *Twisted Ladder Topology* with 8 nodes and 64 processing units.

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, 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. Figure 2.2 shows a snippet of a server which is plotted with the tool *lstopo* from the *hwloc library* [25]. The snippet shows one CPU socket (Package #0) divided into two NUMA-nodes (NUMANode #0 and NUMANode #1). Each node consists of 8 PUs, with a private L1 data cache (L1d) and a shared L1 instruction cache (L1i). The L2-caches are shared by two PUs and stores data as well as instructions. Finally, the L3-cache is the last level cache (LLC) and is shared among a whole NUMA-node.

Data in caches is stored in granularity of cache lines. A cache line in a commodity hardware often has a size of 64 bytes and can hold multiple independent values. When a PU requests data from a specific address the caches are checked for the corresponding data first. A *cache hit* signifies that the data has been found, otherwise a *cache miss* occurred and the data must be fetched from the main memory. It is possible that multiple PUs work on the same set of data and several caches hold a copy of the same address. While reading from the same data location has no side effects, writing to it invalidates data in other caches. Memory architectures of commodity hardware provide a *cache coherence* mechanism to prevent PUs from working on invalid data. Cache coherence algorithms track invalidations of cache lines and propagate them to all other copies in order to refresh invalid cache lines on access.

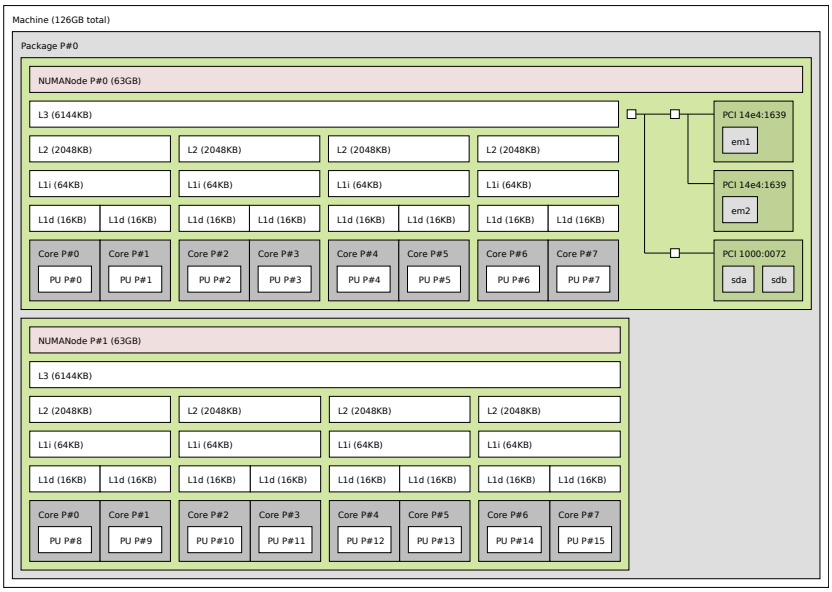


Figure 2.2: The cache hierarchy of a CPU socket with two NUMA-nodes of a server (generated with hwloc [25]).

2.4 Data Locality

The *Principle of Locality* [30], also known as the *Locality of Reference*, is a fundamental principle of computer science. It describes a phenomenon in which software uses specific values or memory regions more often than others. This allows to predict the behavior of software and enables performance optimization techniques like caching and prefetching of data.

> [Locality can be defined] "in terms of a distance from a processor to object x at time t, denoted $D(x,t)$. Object x is in the locality set at time t if the distance is less than a threshold T: $D(x,t) \leq T$." [30]

The distance D can be temporal or spatial. *Temporal locality* is the duration between the reuse of the same piece of data. The duration can be caused by the behavior of the software or the delay on a network link. *Spatial Locality* is a topological distance (physical or virtual) between pieces of data. In the context of a NUMA architecture distances can be measured in the number of hops required by a PU to access a memory region. Further examples for types of localities are *Instruction Locality* [31], *Branch Locality* [32] and *Sequential Locality* [33]. A more general term without a specification of the actual metric is referred to as *Data Locality*.

Knowledge about data locality can be exploited by hardware and software to increase the overall performance of a system. Often used hardware mechanisms to improve data locality

are caching and prefetching of data. Caching relies on the idea that recently used data has a high probability to be requested again in the near future [34] (temporal locality). Moreover, it is likely that adjacent data will be requested, too [34] (spatial locality). This encourages to prefetch data and store it proactively in caches. This technique is called cache prefetching and is designed in order to reduce the number of accesses to slower memory and to reduce related access latencies. Table 2.1 gives an overview of load access latencies which is not complete and applicable to all systems. It shows measured CPU cycles per memory access type and was measured by David et al. [35] on a system with 4x *AMD Opteron 6172* processors with *HyperTransport 3.0* and on a system with 8 *Intel Xeon E7-8867L* with *QuickPath Interconnect*. The table illustrates that the distance between PU and memory location has an impact on the access latency by up to two order of magnitudes and subsequently an impact on the performance of software.

Memory Load Type	AMD Opteron 6172 6.4GT/s HyperTransport 3.0	Intel Xeon E7-8867L 6,4GT/s QuickPath
L1	3	5
L2	15	11
L3	40	44
Local RAM	136	355
Remote RAM (one hop)	247	492
Remote RAM (two hops)	327	601

Table 2.1: An overview of memory access latencies in CPU cycles [35].

The data locality in software can be increased by adjusting the memory placement and the memory access pattern. Compacting data to the NUMA-node of the executing task avoids remote memory accesses and placing data linearly according to the access sequence supports hardware prefetching (sequential locality). Listing 2.1 shows two slightly different versions of a function which summarizes all elements in a matrix. At first, the matrix of the dimension $SIZE * SIZE$ is allocated and filled. To summarize all elements in the matrix, two nested for-loops generate all combinations of the indexes i and j to cover all elements of the matrix, where i is the running index of the outer loop and j is the index for the inner loop. In the first version sum is calculated by adding the elements column-wise which bypasses the prefetch mechanisms and causes a high number of cache misses, thus leading to poor performance. In the second version sum is calculated line-wise and benefits from an access pattern which leverages the sequential locality.

```
1 int matrix[SIZE][SIZE] = initalize_and_fill_data();
2 int sum = 0;
3 for (int i = 0; i < SIZE; ++i) {
4   for (int j = 0; j < SIZE; ++j) {
5     sum += matrix[j][i]; // (version 1) SLOW - column-wise access
6     sum += matrix[i][j]; // (version 2) FAST - line-wise access
7   }
8 }
```

Listing 2.1: Example of a locality optimized memory access pattern

The Linux operation system is NUMA-aware and provides tools for memory placement and scheduling optimization [36, 37]. Memory can either be allocated explicitly from specific nodes or via process and system wide polices. Linux supports the allocation polices *first-touch* and *interleave*. The first-touch policy allocates data at the NUMA-node where the process or thread (task) is currently executed. This is the default policy on most Linux systems and promises an optimal data placement for small programs. The Linux scheduler tries to maintain the node of the currently running task to obtain the data locality. If a program allocates more memory than available on the current node, it falls back to memory of a nearby node. The scheduler can move tasks to other nodes to balance workload. To prevent the scheduler from moving tasks to other nodes, the tasks can be manually pinned to a set of PUs [38]. Memory pages can be migrated between nodes [39] to improve the data locality. The pages which shall be migrated are marked and will be migrated on next access (lazy migration). The interleave policy allocates the memory pages in a round robin fashion from all nodes to reduce worst case memory access scenarios.

Optimizing the data locality can enhance performance, but can also lead to a degradation. False Sharing [40] is an effect that can occur in multi-threaded applications when a cache line is shared among multiple caches. In detail, two objects A and B can be located next to each other in memory and fit into the same cache line. The thread T_A operates only on object A and T_B operates only on object B. Running concurrently, the cache line is shared among the caches serving threads T_A and T_B. In this case, T_A and T_B do not share any data from the logical view, but from the perspective of the memory architecture they do. This conflict leads to a degradation in performance when one of the thread starts to update its object. Then, the other thread has to reload the cache line on access, even if its object did not change.

2.5 Scheduling

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 PUs (computing resources). Scheduling algorithms also called scheduling disciplines [41] may focus on specific aspects like fair sharing of all resources, maximizing the throughput or minimizing response time. We are looking for a scheduling discipline which takes the data locality into account to improve the performance of applications.

CAF provides the scheduling disciplines work-sharing and work-stealing [42]. Work-sharing is usually a First Come, First Serve (FCFS) scheduler [43] which has a centralized job queue. Work items are enqueued at the tail and workers dequeue items from the head and execute them. This can cause contention due to the synchronization requirements. Work-stealing reduces this contention by introducing one job queue per worker. Each worker operates on its own job queue until it is drained. Then, it picks another worker and tries to steal a work item. The victim is chosen at random to avoid any kind of bias and to reduce the amount of required information to the number of workers to steal a job.

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. An unfair utilization can lead to the convoy effect [44] where a long running job blocks other jobs from execution and reduces the throughput temporally.

A *work-conserving scheduler* tries to keep every worker busy while a non-work-conserving scheduler [45] can withhold jobs on purpose to reduce the workload of workers. Such a strategy could allow to reserve computational and bandwidth resources for tasks with higher priorities or allows to prevent the performance degradation pattern *trashing* [46]. Originally, trashing refers to the swapping of pages between main memory and hard disk. Other flavors exist like cache-trashing which leads to a high number of cache misses. In a NUMA-system, a non-work-conserving strategy can be used to reduce the contention on interconnection links. Further scheduling characteristics are discussed in Section 3.2.

A typical use case for task and actor based programs is the divide and conquer paradigm where work items can generate new work items to split problems into smaller problems.

Consequently, each work item can have multiple children and at least one parents with the exception of the first work item. A work item which requires multiple preconditions for its execution might have multiple parents. These relationships are directed and acyclic and allow to form a directed acyclic graph (DAG), where nodes represent work items and edges specify their relationships. In task based parallelism, work items can be represented as tasks and are the scheduled entities. This differs from the actor model wherein work items can be represented as messages while actors are scheduled.

Whether a DAG is deterministic in respect to the program input, and whether it can be generated ahead of the program execution depends on the concurrency model and its implementation. In the case of CAF, the actor pattern is by its nature of asynchronous message passing non-deterministic. The DAG can only be generated dynamically at runtime, because CAF has no knowledge about the behavior of the application. A DAG can be traversed by the scheduler in a *breadth first search* (BFS), in a *depth first search* (DFS) policy [47], or in a combination of it. The time complexity is the same for both polices when all edges have to be traversed. When a program can be successfully completed without traversing all edges, the best choice depends on the program. The total amount of required memory (space complexity) can be much better for DFS than for BFS. For instance, a DAG in the shape of a balanced tree is given, where b is the number of branches at each node and h is the height of the tree. Then, DFS has a space complexity of $O(h)$ and BFS has a complexity of $O(b^h)$. DFS can be implemented by adding new jobs to the head of the job queue to simulate the behavior of a stack (LIFO), while BFS adds new jobs to the tail (FIFO). On the one side, processing jobs in a LIFO fashion reduces the memory footprint and increases cache locality. On the other side, DFS cannot be parallelized. To parallelize a program, BFS can be used. It traverses a tree in a level order and creates all work items for the next level which can be executed concurrently. DFS and BFS can be combined by adding work items in an arbitrary ratio to the head and to the tail of the job queue.

The overall runtime of an application which is also called makespan in the context of scheduling can be counter-intuitively increased by optimizing software or upgrading hardware [48, 49]. Such anomalies can be caused by the scheduling on multicore systems after reducing the execution times of individual jobs, weakening dependencies or increasing the number of PUs.

The following example illustrates three anomalies. Figure 2.3 shows a DAG of a test program with the jobs J0 as root and J1 to J7 as children or grand children. Each job has a makespan of t time units. To successfully complete this program all, jobs have to be executed. For simplicity the scheduler uses work-sharing with a central and pure FIFO job queue. Jobs are created at

the end of other jobs and jobs which are created in the same time slot are added to the queue ordered by their index starting with the lowest. The host system consists of either two or three processing units P1, P2 and P3. At the beginning of each time slot, idle PUs sequentially fetch jobs from the job queue in order of their index. At program start, J0 is the only job in the job queue. Figure 2.4(a) shows the baseline with two PUs and an optimal t of 8. The following list describes the process time slot by time slot:

1. At program start P1 fetches the only job in the job queue J0 and P2 stays idle. During execution, J0 spawns three jobs J1, J2 and J3 which are added in the same order to the job queue.

2. P1 fetches J1 and P2 fetches J2.

3. While P1 is still busy with J1, P2 fetches J3 (the last job in the queue). At the end of this time slot J1 creates J7.

4. P1 fetches the just added job J7 and executes it up to the time slot 8. P2 is still busy and creates the jobs J4, J5 and J6 which are executed sequentially in the time slots 5 to 7 by P2.

In the baseline scenario, the critical execution path is scheduled optimally with the jobs J0, J1 and J7 in a sequence. The three scheduling anomaly examples follow the pattern and disrupt its critical path with a non critical job which leads to an increase of the makespan. The scheduler cannot prevent such a disruption, because it has no knowledge about the application logic.

Figure 2.4(b) shows an example where an upgrade of the host system by an additional PU increases the makespan by 1 unit. In this case, J1, J2 and J3 can be are executed concurrently in time slot 2. At the end of this slot the jobs J4 to J7 are created and enqueued to the job queue in the same order. Consequently, J7 is the last job in the job queue which leads to an interrupt of the critical path and increases the makespan.

Figure 2.4(c) shows a scenario where the computational cost of job J2 could be saved. This leads again to the sequence of J4 to J7 in the job queue, as a consequence J7 is executed last which disrupts the critical path.

Finally, in Figure 2.4(d) the dependency of J4 is reduced and is created directly from the root J0 instead of from J3. J1 to J4 are created by J0 and enqueued in the same order. In the next two time slots, J1 to J3 are executed and J7, J5 and J6 are added to the job queue. At the beginning of time slot 4, the jobs J4, J7, J5 and J6 are stored by the job queue. This leads to the disruption of the critical path, because P1 fetches J4 and P2 is still busy with J3.

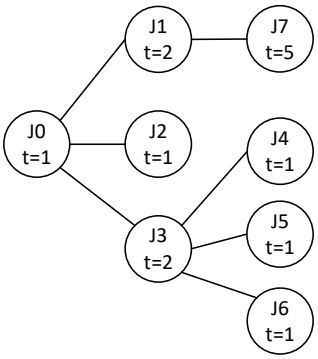


Figure 2.3: Directed acyclic graph (DAG) of the scheduling anomaly example program.

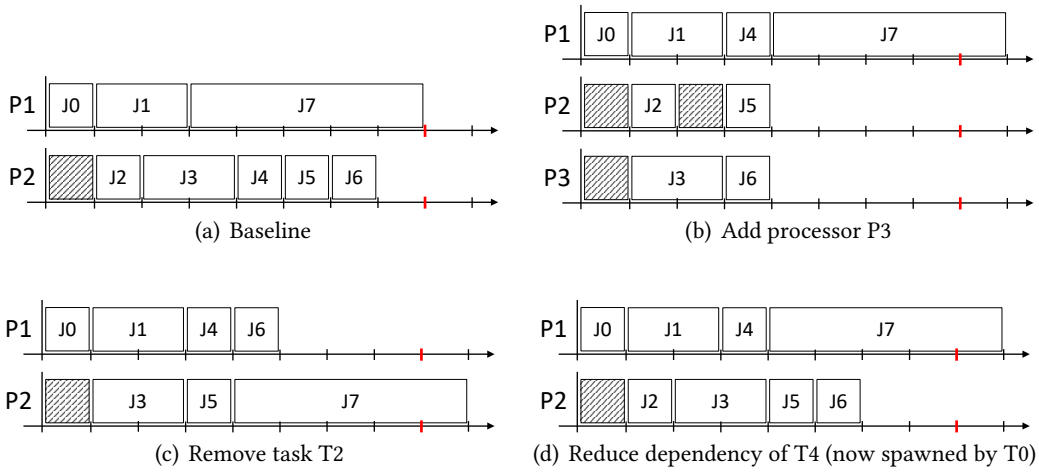


Figure 2.4: shows the baseline scheduling example and three program or hardware optimizations which cause different scheduling anomalies, resulting of an extended makespan.

Approaches exist to prevent problems caused by scheduling anomalies in real time scheduling. For example, Brandt et al. [50] proofs that preemptive EDF (earliest deadline first scheduling) has no anomalies. Another way is to take the upper bound of scheduling anomalies into account and adjust the deadlines and system resources accordingly [48, 51]. A prove for a general upper bound is introduced by R. L. Graham [48]. This theoretical upper bound can be reduced when certain assumptions can be made like jobs have no dependencies [48]. Scheduling anomalies can occur in CAF when optimizing the data locality. However, these are problems which have to be considered when designing hard realtime systems where it musst be ensured that deadlines are never missed. CAF follows the best effort principle, hence single deadlines are irrelevant. The general upper bound is very pessimistic and can usually be neglected.

3 The Problem of Actor Scheduling and Related Work

We are looking for a scheduler which is aware of the memory architecture and considers the non-uniform access characteristics to improve the performance of actor-based applications. With this aim, we survey the current CAF scheduler in detail first. Next, we explore the design constrains of a NUMA-aware scheduler which are imposed by CAF. Finally, we examine the data locality aspects of the actor pattern in respect to scheduling and study the related work.

3.1 Scheduling in CAF

CAF supports the basic actor types `blocking_actor` and `scheduled_actor`. The former is scheduled preemptively by the operating system, while the latter is scheduled cooperatively in the user-space by the actor system. The actor system executes scheduled actors in a thread pool, whereas the number of threads are preconfigured and by default are bound to the number of cores of the host system. The number of actors depends on the application and can vary over time. Blocking actors are assigned to a dedicated system thread that allows the use of blocking calls like I/O operations which could starve other actors. However, this practice should be used with caution, because spawning an actor in a dedicated thread is an expensive operation.

A scheduled actor can be in one of the four states (1) Waiting, (2) Ready, (3) Running or (4) Done as shown in Figure 3.1. An actor is either spawned lazily or eagerly. By default an actor is spawned eagerly and starts in state Ready to run its initialization as soon as possible. During the initialization process, an actor can adjust its behavior and execute code, e.g., to prepare future calculations by starting actors or sending messages. A lazily spawned actor delays its initialization and stays in state Waiting until the first message is received. On receipt, an actor in state Waiting switches to state Ready and is scheduled for execution. The first time an actor is executed it performs its initialization. Then, it starts processing messages sequentially by taking them from its inbox up to a preconfigured number. If the maximum number of messages per schedule is reached, the actor switches either to state Waiting or back to state Ready. Defining a high number of messages to process in one schedule, maximizes the overall

number of instructions performed per second by the actor system, whereas a low number minimizes the response time. This approach has the downside that the execution time of an actor is not considered and actors with long execution times can reduce the throughput of the worker. An alternative approach could measure the execution time and could reschedule when a threshold is exceeded instead of processing the next message. This increases the fairness between actors, but also increases the computational effort on a critical code path. Calling the function `quit()` closes the inbox of an actor, releases its resources and switches to the final state Done. A one-shot actor, i.e., an actor spawned without a behavior, cannot receive messages and therefore never switches to state Waiting and quits automatically after execution.

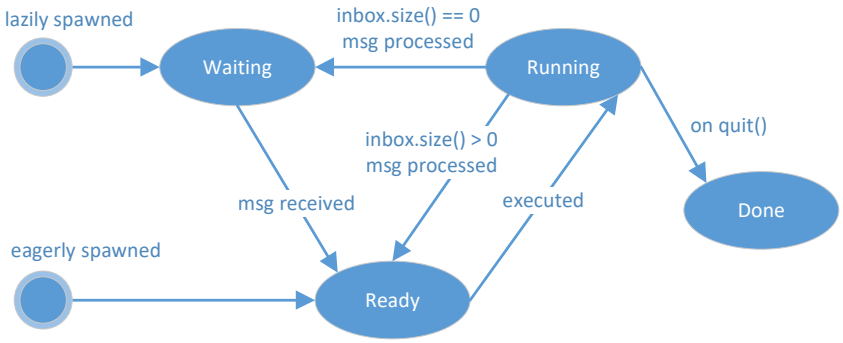


Figure 3.1: A scheduled actor can be in one of the states: Waiting, Running, Ready, Done.

The cooperative scheduling algorithms work-sharing and work-stealing provided by CAF are implemented as follows. The work-sharing scheduler has a centralized job queue protected by a mutex with a condition variable according to the bounded buffer problem [52]. In this problem, consumers acquire data items from a data structure while producers add new items. The data structure synchronizes accesses of consumers and producers. If the data structure is empty it blocks consumers until new items are added and it blocks producers when the data structure is full. While a job queue has no capacity limit, it can be empty. In contrast work-stealing uses a job queue per worker whereby a job queue is not allowed to block workers. Instead a worker has to poll other job queues and decide himself which frequency to use and when to take a break. It aims a trade-off between response time and CPU usage. CAF distinguishes between three polling strategies: aggressive, moderate and relaxed. At first, a drained worker tries to steal a work item by aggressively polling other workers job queue to acquire a new job as fast as possible to minimize its idle time. After a preconfigured number of unsuccessful tries, the worker falls back to the moderate stealing strategy. The polling frequency is reduced by short intermediate sleep intervals to stay responsive while the contention on other workers job queue is reduced. Finally, after a number of moderate tries, the worker falls back to the

relaxed strategy. The polling frequency is further reduced to minimize the CPU load, when no work is available. After a successful steal the stealing strategy is reseted to aggressive.

Work items are added to workers job queue via the functions `internal_enqueue()` and `external_enqueue()`. `Internal_enqueue()` is only called by its own worker and enqueues work items at the head to schedule the actor to be executed next. The function is called by the current running actor for newly spawned actors and for idle actors which received a message. Queue-jumping (enqueuing at the head instead of at the tail) is done to increases the temporal locality, because newly spawned actors or sent messages might still be cached when the next actor is executed. `External_enqueue()` is called by other workers and enqueues work items at the tail to avoid interference with local optimization. For the same reason, work-stealing dequeues work items from the tail. To interact with blocking actors and to handle work items from a non-actor context, workers are managed by a *Coordinator* which provides the function `central_enqueue()` to schedule external work items. On enqueuing, the function chooses a worker and calls its `external_enqueue()` function while the worker is chosen in a round robin fashion to balance the workload.

3.2 Scheduling Constraints

A scheduler for a user-space actor system such as implemented in CAF 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 makespan of an application. Finding the best scheduling decision is a well-known optimization problem called the *Job Shop Scheduling Problem* [53]. It describes the challenge of assigning a number of work items of varying execution times to a number of PUs. Finding the optimal solution is an NP-hard problem. Job Shop Scheduling can be distinguished between static and dynamic scheduling. Static scheduling is an offline problem and information such as the number of work items, their dependencies, and individual execution times are provided upfront. In contrast, the CAF scheduler attempts to solve an online problem where work items are generated dynamically and individual execution times are unknown. However, approaches to estimate execution times exist. For example, Wang et al. [54] predicts the execution time in a task based parallelism framework based on previous execution of tasks which can be adapted to the actor model.

Prioritizing performance-critical work items such as items that have many dependencies to future work is a strategy to reduce the makespan [55]. 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 [56]. 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 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.

3.3 The Problems of Locality

A scheduler can optimize the communication effort between actors, called communication locality (CL) [5], or the efforts of a PU of accessing the data by its executing actor, known as execution locality (EL) [6]. 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 the 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 is access characteristics and the communication overhead. CAF cannot solve this challenge without the 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 references a large data structure to be processed by the recipient. Consequently, we need to consider different metrics.

CAF can adjust CL and EL by scheduling decisions at two opportunities: (1) a worker finished its job and is looking for new work, or (2) an idle actor receives a message. CAF uses random work-stealing to acquire new work for idle workers. Although very simple, this strategy ignores data locality and misses opportunities to improve the application performance. An actor without a message in its mailbox is considered idle and is not enqueued in any job queue. 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.

3.4 Related Work

Work-stealing is widely used for scheduling actors or tasks for example by OpenMP [57], Erlang [19], Akka [12], the Pony Language [18] and CAF [3].

Random work-stealing (RWS) [4] scales well by following a distributed approach, it is *stable* [58], because it requires little action 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) [59]. 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 a remote cluster is much more likely due to the (uniform) randomness when choosing a victim.

We experience a similar problem within a NUMA machine. Here, work may be 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 [59]. However, none of these solutions are feasible for a NUMA-aware scheduler, because they hide high network delays by prefetching mechanisms instead of improving the data locality.

Scheduling algorithms such as work-sharing or random work-pushing [58] 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 an increase of unsuccessful push attempts that degrade performance.

There are several approaches to work-stealing that consider data locality. Acar et al. [60] 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 an affinity for a specific PU to a thread and equipping workers with a priority 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 costlier for actors than for threads due to the much higher quantity of actors.

Olivier et al. [57] propose a hierarchical scheduling approach by combining work-stealing with work-sharing to exploit shared caches and improve the data locality in NUMA-systems. 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.

Wang et al. [54] use a class-based scheduling approach to categorize tasks based on their memory footprint. 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 task is chosen based on factors like data size and the expected execution time. Acquiring the memory footprint of an actor or message is only possible with the support of the application developer which we want to avoid in CAF.

Quintin et al. [61] 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 variant occupancy-based stealing is proposed by Contreras et al. [62]. It chooses the victim by the largest job queue to balance the workload. Hence, this requires to probe all victims, they are divided into groups to mitigate the overhead. The thief choses a group at random and the victim in the group with occupancy-based stealing. This approach reduces the number of steal attempts for applications which tends to lead to an imbalanced workload.

The actor communication patter of *hubs* and *hub affinity groups* was introduced by Francesquini et al. [63] 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 the PWS [61] 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 take it into account when scheduling actors 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.

4 Design of 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.

4.1 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 [61]. 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.

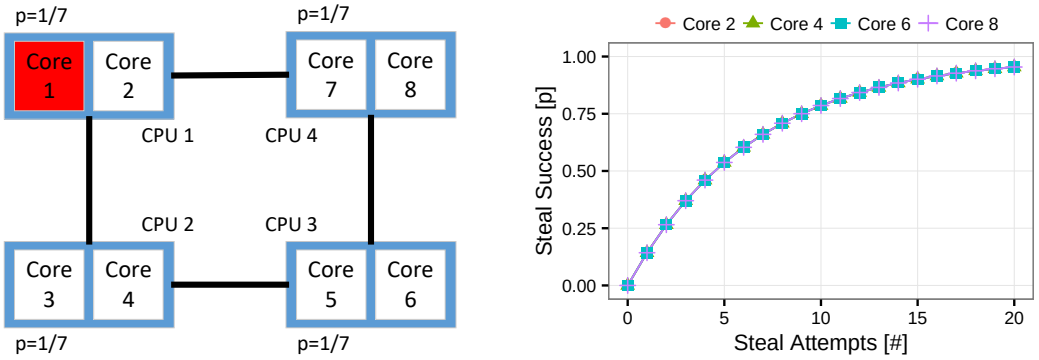
Once a worker runs out of work, it becomes a thief and tries to steal work items from all victim groups in increasing order. The steal attempts per group depend on the size of the group. The thief performs the lowest number of steal attempts on g_0, but picks victims from the final group g_k indefinitely. 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 of processing 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 [63], 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 mechanisms. 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 4.1(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 4.1(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 4.2 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 4.2(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

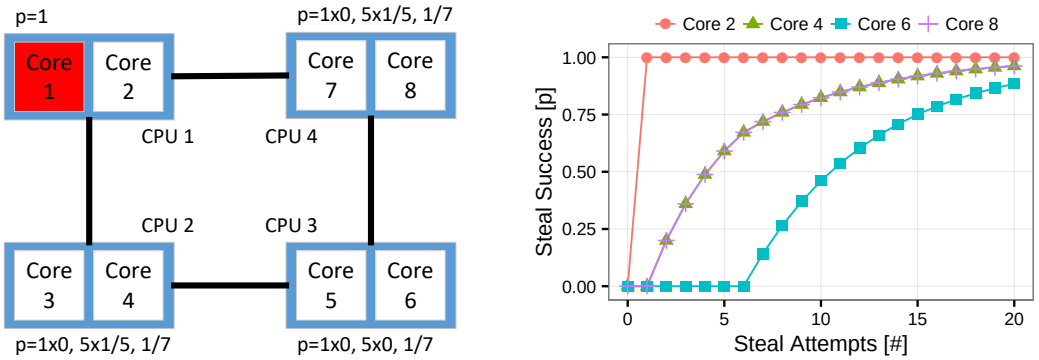
(b) Probability vs. steal attempts

Figure 4.1: Chances for successfully stealing a work item from core one using *random* work-stealing.

4.2 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 by our scheduler is static soft pinning that is automatically handled by the framework.

Similar to the approach of Francesquini et al. [63], 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



(a) Core layout with probabilities

(b) Probability vs. steal attempts

Figure 4.2: Chances for successfully stealing a work item from core one using *locality-aware* work-stealing.

worker persistently as its home processing unit (HPU) on first execution as proposed by Acar et al. [60]. 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 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 4.3 shows where an idle actor is enqueued when it receives a message.

This approach requires no additional effort from 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.

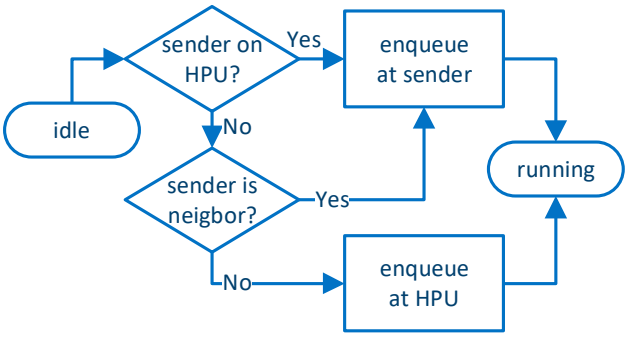


Figure 4.3: Flowchart to determine the worker during the scheduling phase.

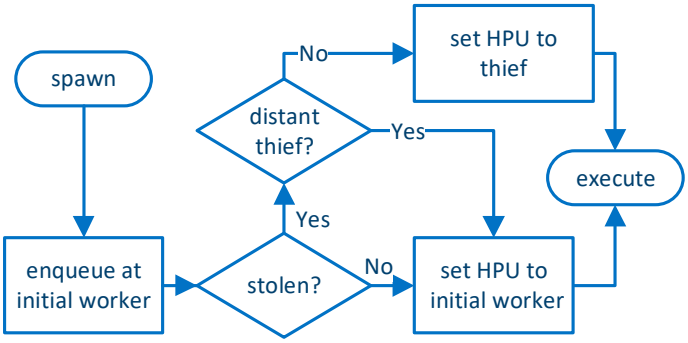


Figure 4.4: Flowchart to determine the HPU during actor initialization.

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 [54] or a soft [60] 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 [63] 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 [60] does not require 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.

4.3 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 6.5.2 displays a scenario where this behavior impacts performance: a system hosts two actors that exchange messages in a ping-pong pattern. Both actors are placed at the same worker when spawned and immediately scheduled for execution to initialize 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 do not 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 4.4. 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.

5 Implementation

In this chapter, we shortly introduce the library hwloc which we use to gather and exploit hardware information. Then, we give an overview on how the scheduler in CAF is designed and describe the implementation of LGS in detail.

5.1 Hwloc

Portable Hardware Locality (hwloc) [25] is a library of the Open MPI project[1]. Hwloc gathers information about the hardware topology of the host system including the NUMA architecture, CPU sockets, caches and I/O devices. It provides a convenient API to access these information and to exploit it.

For example, after initializing hwloc and loading the hardware topology a list of all PUs and Nodes can be accessed by the functions `hwloc_topology_get_complete_cpuset` and `hwloc_topology_get_complete_nodeset`. A matrix which stores the distance between all NUMA-nodes is returned by this function `hwloc_get_whole_distance_matrix` and the root object of the hardware topology which is structured as a tree by the function `hwloc_get_root_obj`. The tree can be traversed, whereby a node represents either a CPU socket, a cache level or a PU. Additionally, a node stores information like the cache size. These information can be exploited by binding the current running thread to a set of PUs with the function `hwloc_set_cpubind` and a thread can be forced to allocate memory from a specific NUMA-node with `hwloc_set_membind`.

5.2 The Architecture of the Actor Scheduler in CAF

In this section, we give an overview of the software architecture of CAF with a focus on the scheduler. Figure 5.1 shows relevant classes and their relationships. The class diagram is simplified and omits several functions and interfaces. The class `coordinator` loads a number of `workers` on startup. Each worker creates a thread to process work items (*resumable*)

[1] https://www.open-mpi.org/

concurrently. The scheduling algorithm is defined separately from the worker which allows to simply exchange the behavior of the workers. A policy based design is used to equip a worker with a specific scheduling algorithm.

The coordinator is started by the `actor_system` which represents the environment for actors. The actor system loads on startup modules like I/O abstraction, actor registry and a configuration manager (`actor_system_config`) which allows to fine-tune CAF. The `actor_system_config` can be used to specify a scheduler as well as to configure scheduler internals like the number of workers and polling intervals.

5.3 The Locality-Guided Scheduling

The implementation of the Locality-Guided Scheduler (LGS) consists of an initialization part, weighted work-stealing and soft actor pinning. In the initialization, LGS gathers hardware information and pins worker threads to specific PUs. The initialization and weighted work-stealing are implemented as the policy *locality_guided_scheduling* and inherit most functions from the class `work_stealing`. However, actor pinning is implemented in the actor class `scheduled_actor` and only controlled by the policy LGS.

Figure 5.1 shows the design of the scheduler where modifications and additions for the implementation of LGS are marked red. We extend the worker class by adding setter and getter functions for direct neighbors and we updated the function `exec_later`. `Exec_later` enqueues work items (resumables) to the job queue and the added flag allows to specify if the work item has to be added at the tail or at the head. We further extend the scheduling policies by the function `init_worker_thread` which is directly called after a worker has started its thread. While this function is empty for the classes `work_stealing` and `work_sharing`, LGS uses it for pinning the threads to PUs.

Listing 5.1 shows the initialization of worker threads by LGS. First, we define `wdata` and `cdata` to enable access to the context of controller and worker. Next, we pin the current thread to the PU which has the id of the worker. Subsequently, we gather hardware information and sort all other PUs by their distance and store the result in a matrix with the function `init_worker_proximity_matrix`. The first row in the matrix (`wp_matrix[0]`) contains all direct neighbors which is passed to the worker.

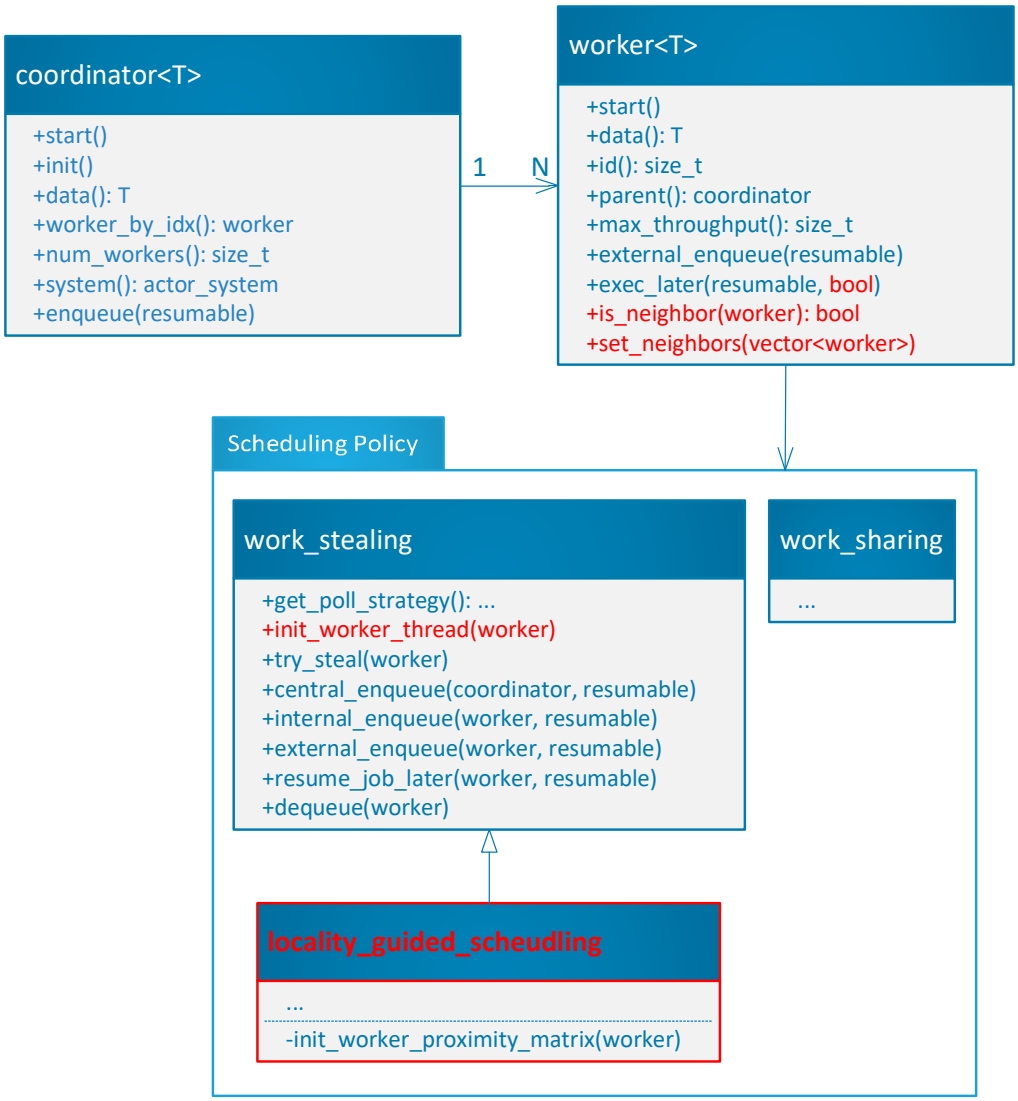


Figure 5.1: A Coordinator manages a number of workers, whereby the scheduling algorithm is defined separately and can be exchanged. Functions and classes we added or changed are marked red.

```
1  template <class Worker>
2  void init_worker_thread(Worker* self) {
3    auto& wdata = d(self);           // data of worker
4    auto& cdata = d(self->parent()); // data of controller
5    // pin current thread to PU
6    auto pu = hwloc_bitmap_make_wrapper();
7    hwloc_bitmap_set(pu.get(), self->id());
8    auto res = hwloc_set_cpubind(cdata.topo.get(), pu.get(),
9                 HWLOC_CPUBIND_THREAD | HWLOC_CPUBIND_NOMEMBIND);
10   CAF_ASSERT(res == -1);
11   // get distance matrix
12   wdata.wp_matrix = wdata.init_worker_proximity_matrix(self, pu);
13   ...
14   self->set_neighbors(wp_matrix[0]);
15   ...
16 }
```

Listing 5.1: Initalization of worker threads

Listing 5.2 shows the implementation of determining the home processing unit (HPU) for an actor. It follows the design shown in Figure 4.4. The parameter `eu` points to the current worker which wants to execute the actor and `hpu_` is a private member variable of the actor storing the HPU. In the process of spawning an actor, the HPU is set to the current worker (the initial worker). When a worker executes an actor, the function `activate` of the actor is called. When the actor is executed for the first time, it is initialized. One-shot actors are finalized afterwards and never executed again. Other actors set their HPU after the initialization accordingly.

```
1  bool scheduled_actor::activate(execution_unit* eu) {
2    ...
3    if (!getf(is_initialized_flag)) {
4      initialize();
5      if (finalize()) {
6        ...
7      } else {
8        if (hpu_ != eu) {
9          // actor was stollen after spawning
10         if (hpu_ == system().dummy_execution_unit()
11             || hpu_->is_neighbor(eu)) {
12           hpu_ = eu; // set the HPU to the current PU
13         }
14       }
15     }
16   ...
17 }
```

Listing 5.2: The HPU is set to the current PU under specific circumstances

Listing 5.3 shows the implementation of how actors are pinned to their HPU. It follows the design shown in Figure 4.3. When an actor receives a message the function `enqueue` is called. It adds the message (`mailbox_element`) to the inbox of the actor. This might return the status `success` or `unblocked_reader`. The latter indicates a successful enqueue, but an idle actor. Idle actors must be re-scheduled and enqueued to a job queue of a worker. The actor can either be enqueued at the HPU or at the current worker (`eu`). Enqueuing of actors to job queues are further distinguished between external enqueue and internal enqueue. The former enqueues the actor at the tail while the latter is enqueued at the head.

```
1  void scheduled_actor::enqueue(mailbox_element_ptr ptr,
       execution_unit* eu) {
2    ...
3    // add message to mailbox
4    switch (mailbox().enqueue(ptr) {
5      // successfully enqueued
6      case detail::enqueue_result::success:
7        ...
8      // actor is idle and needs to be re-scheduled
9      case detail::enqueue_result::unblocked_reader: {
10       ...
11       if (eu) {
12         // msg is received from an other scheduled actor
13         if (eu == hpu_ || eu->is_neighbor(hpu_)) {
14           eu->exec_later(this, true); // internal enqueue
15         } else {
16           // ‘eu‘ has a high memory distance to this actor
17           hpu_->exec_later(this, false); // external enqueued
18         }
19       } else {
20         // msg is received from a non-actor context or from a
               detached actor
21         hpu_->exec_later(this, false); // external enqueue
22       }
23     }
24   }
25   ...
26 }
```

Listing 5.3: Actors are pinned to their HPU

6 Evaluation

In this chapter, we evaluate the scheduling discipline Locality-Guided Scheduling (LGS). We examine how LGS affects the data locality and the performance. Subsequently, we identify scheduling problems and discuss possible solutions.

In Section 6.1, we present our test server and the tools we use for our evaluation. Next, in Section 6.2, we describe the popular actor benchmark suite Savina and our self designed benchmark Matrix Search. In Section 6.3, we describe our first experience with LGS and Matrix Search. We summarize further analysis and findings related to Matrix Search in Section 6.4. Finally in Section 6.5, we study LGS with the Savina benchmark suite and analyze specific benchmarks which stand out.

6.1 Measurement Setup

6.1.1 Test Server

All measurements were performed on a *Dell PowerEdge R815* server with four *AMD Opteron 6376* processors, clocked at 2.3 GHz. Figure 6.1 shows the hierarchical memory architecture of our test server created by the hwloc utility [25] *lstopo*. A CPU socket is represented by a `Package` which is divided into two `NUMANodes`, 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 with 512 GB of main memory. Each core has its private L1-data cache. The L2-cache is shared by two cores and the L3-cache is shared by all cores of a NUMA-node. The NUMA-nodes are connected via bidirectional, point-to-point and cache coherent links using the *HyperTransport* [64] technology. The structure of the interconnection network is shown by a *System Locality Distance Information Table* (SLIT) [29] in Table 6.1. It shows the number of hops between two NUMA-nodes. The table is diagonally reflected, because the interconnection link between each two nodes is symmetric. The distance for the local node to itself is zero and the maximum number of hops to reach each node from each other node is limited to two.

Our test server provides the programmable hardware counters Core and Uncore [29]. We use them to analyze performance issues in our benchmarks. The former counts core specific events

like cache misses, TLB misses and instruction prefetching. The latter counts Northbridge specific events like local and remote main memory accesses, I/O accesses and thermal status. These counters have very low runtime overhead, but are limited to 5 Core counters per PU and to 4 Uncore counters per socket. All counters are configurable via the *Maschine Specific Register* (MSR) interface [65].

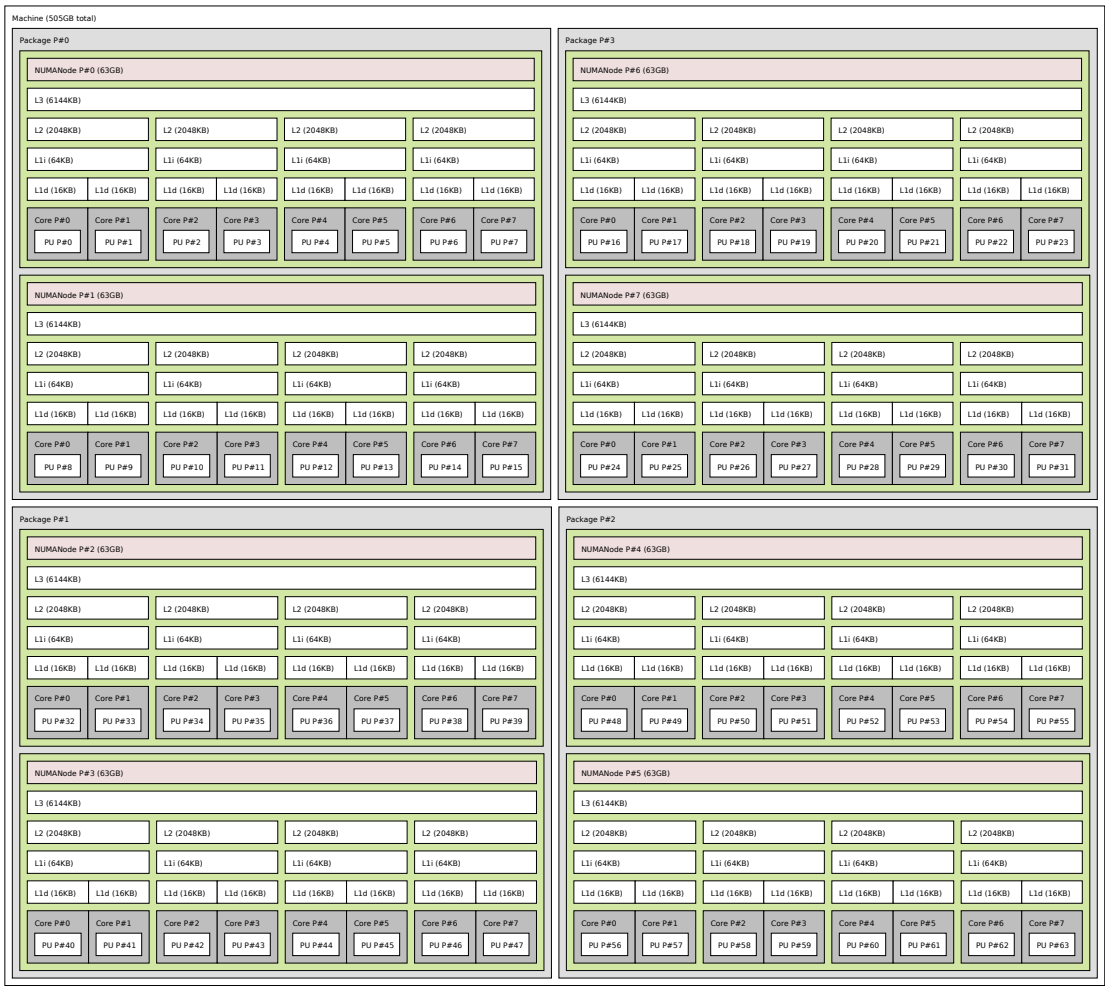


Figure 6.1: Architecture view of our test server generated by hwloc
(Command: `lstopo -no-legend -no-io`)

Node	0	1	2	3	4	5	6	7
0	0	1	1	2	1	2	1	2
1	1	0	2	1	1	2	2	1
2	1	2	0	1	1	1	1	1
3	2	1	1	0	1	1	2	2
4	1	1	1	1	0	1	1	2
5	2	2	1	1	1	0	2	1
6	1	2	1	2	1	2	0	1
7	2	1	1	2	2	1	1	0

Table 6.1: NUMA-node distance matrix of our test server (Command: `hwloc-distance`)

Software	Version
Linux Kernel	3.16.7
GCC	4.8.3
Java	1.8.0
Hwloc	1.11.8
Likwid	4.2.1
Perf	3.16.4
CAF	0.14.6

Table 6.2: Software, tool and library versions used for our measurements

6.1.2 Tools and Libraries

Our test server is operated by a Linux distribution with default NUMA-settings (first-touch). Further software, tools and libraries that we use are summarized in Table 6.2. GCC (*GNU Compiler Collection*) is used to compile CAF and the benchmarks. The Hwloc (*Hardware Locality*) [25] library provides a convenient interface to obtain information about the hardware topology. Likwid [66] is a performance monitoring and benchmark suite we use to configure and read hardware counters. Perf is a Linux profiling tool which allows a detailed performance analysis of applications. It uses statistical sampling and is able to record various software counters like page faults and context switches. Finally, we use Perf as the backend for *Flame Graphs* [67] which is a handy way to visualize stack traces and allows to see which functions in a program consume the most processing time.

All measurements are done with the CAF benchmark suite. It consists of a number of scripts and tools to automatize preparation and execution of benchmarks, editing of data and plotting of graphs.

We measure each benchmark with 4 to 64 workers. 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.

Each benchmark reads a configuration file on startup which allows to adjust the behavior of the scheduler and CAF internals. Furthermore, a predefined set of parameters are passed on startup to the benchmarks and may allow to adjust the level of concurrency, the problem complexity, and other characteristics.

Each measurement is repeated 10 times which leads to 160 measuring points per graph. After execution, the collected raw data is structured and plotted with the statistical programming language *R*. A graph displays the mean as well as error bars that show the 95% confidence interval.

6.2 Benchmark Description

6.2.1 The Savina Benchmark Suite

Savina [7] is a comprehensive benchmark suite for actor libraries with the goal to cover a wide range of concurrency patterns. It currently consists of 30 benchmarks, including micro-benchmarks like *Ping-Pong* and *Recursive Fibonacci*, classical concurrency problems like *Dining Philosophers* and *Sleeping Barber*, and various applications of different styles of parallelism like *Quicksort* and the *N-Queens Problem*. All benchmarks can be configured and allow to adjust the problem complexity and the degree of concurrency. Savina currently supports nine JVM-based actor libraries and includes among others Akka [12], Habanero-Java library [13], Jetlang [14] and Scalaz [15].

We translated 23 of the 30 Savina benchmarks to CAF[1]. In detail, we updated the benchmark suite and all benchmarks in order to improve the comparability to C++. For example, the original Savina benchmarks use a pseudo random generator from the standard Java library which is not comparable to the one in the C++ library. We implemented our own and integrated it in the Java and C++ benchmarks. Not all benchmarks could be translated to C++, because

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

some are using Java-specific libraries like *BigDecimal* where a translation is out of scope for this thesis.

We run the benchmarks with our own predefined set of configuration parameters instead of with the default ones. We increase the problem complexity as well as the level of concurrency when possible to expand the execution time to better measure the scheduling impact on our 64 core test server. The following list summarizes each benchmark and provides our configuration. A more detailed description of the benchmarks and their origin can be found in the original paper [7].

Micro-benchmarks are designed to study and compare specific features of actor systems like the messaging or spawning overhead.

1. **Ping-Pong** consists of two actors sending a message back and forth for N times. ($N = 2,000,000$)
2. **Thread Ring** arranges N actors in a ring. A token is passend from one actor to the next one along the ring. The token traverses sequentially two times R number of actors. ($N = 1,200$; $R = 1,200,000$)
3. **Counting Actor** consists of two actors. One sends N messages to the other one, who fetches them from its inbox. ($N = 10,000,000$)
4. **Fork Join (Throughput)** consists of one sending actor and A receiving actors. The sending actor sends one messages to all other actors. This is repeated sequentially N times. On receipt, a small computation is performed. ($N = 60,000$; $A = 360$)
5. **Fork Join (Actor Creation)** creates an actor and sends a message to it. This is sequentially repeated N times. A spawned actor fetches the message from its inbox and performs a small computation before it terminates. ($N = 4,000,000$)
6. **Fibonacci** calculates the Nth Fibonacci number recursively. In each recursion, two actors are spawned and a message is sent to both to calculate the Fibonacci numbers $n - 1$ and $n - 2$, where n is the current Fibonacci number at the current recursion level. ($N = 34$)
7. **Chameneos** arranges meetings by a central actor called *Mall* for two *Chameneos* actors. On meeting, the pair of Chameneoses exchange and update their state. Chameneoses which are not in a meeting ask the Mall to arrange one. The number of Chameneoses C as well as the number of meetings M can be configured. ($C = 4,000$; $M = 800,000$)

8. **Big** is a many-to-many Ping-Pong scenario. W Ping-Pong actors are spawned and it is ensured that each actor knows all other actors. Then, a Ping-Pong actor randomly chooses a buddy, sends a ping message to it and waits for the response. In the meantime, it can respond to other Ping messages with a Pong. After it receives its response it chooses a new buddy. Once all actors have received N Pong messages the program terminates. ($N = 60,000; W = 360$)

Concurrency benchmarks test classical coordination problems.

9. **Concurrent Dictionary** encapsulates a hash-map in an actor and provides a read and a write message interface for it. E workers send M read and write requests to the map actor and wait for a response. The ratio W between read and write requests can be configured. ($E = 100; M = 50,000; W = 10$)
10. **Concurrent Sorted Linked-List** is similar to the *Concurrent Dictionary*, but uses a sorted linked-list. ($E = 20; M = 8,000; W = 10$)
11. **Producer-Consumer with Bounded Buffer** uses a manager actor to coordinate a number of P producer actors and a number of C consumer actors. The number of items a producer creates before it terminates is defined by I. The manager uses a buffer with storage places for B items which is filled by producers and emptied by consumers. In case the buffer is full, the manager delays new orders to the producers, while on an empty buffer, the manager delays responses to consumers. ($B = 75; P = 60; C = 60; I = 1,500$)
12. **Dinning Philosophers** describes a classical synchronization problem, where N philosophers are sitting at a round table, while each wants to eat M times. ($N = 80; M = 40,000$)
15. **Logistic Map Series** calculates the equation $x_{n+1} = rx_n(1 - x_n)$ concurrently for T different rs, whereby S iterations are done per concurrent calculation. The equation describes the growth of a population, where x is the ratio of the current population to the maximum population and r is the ratio of reproduction to starvation. One actor (*series-worker*) is assigned per r which delegates each iteration to a compute actor and waits for the response, whereby each iteration is instructed by a coordinator. The key problem for a series-worker is that the communication between coordinator, series-worker and compute actor follows a synchronous request-response pattern. For each instruction from the coordinator a response from the compute actor must be received before the series-worker can

handle the next instruction, while multiple instructions might already be in the inbox. ($T = 10; S = 25,000; r = 3.46$)

16. **Bank Transaction** focuses on the synchronous request-response pattern similar to the benchmark *Logistic Map Series*. A teller actor generates N bank transaction tasks, each including a sender and a recipient from a pool of A bank account actors. The transaction tasks are transmitted to their senders. The sender forwards the transaction to the recipient which replies with an acknowledgment. In the meantime, the account of the sender is locked from other transactions. ($N = 800,000; A = 16,000$)

Parallelism benchmarks are more realistic applications which take full advantage of multi-core machines.

17. **All-Pairs Shortest Path** is an implementation of the Floyd-Warshall algorithm which searches for the shortest path in a weighted graph. The number of nodes is defined by N, the block size by B and the maximum edge weight by W. ($N = 900; B = 150; W = 300$)

19. **N-Queens Problem** searches for S solutions to place N queens on a $N * N$ sized chess board in a non-attacking position. A central coordinator receives incomplete chess boards (work items) and passes them to one of its W workers in a round robin fashion. A worker receives a work item, identifies the free columns in which a queen can be placed and sends a new work item back to the coordinator for each valid position. A complete N-queen puzzle is reported to the coordinator. ($N = 14; W = 100; S = 1,500,000$)

20. **Recursive Matrix Multiplication** splits two given $N * N$ matrices recursively into smaller blocks up to a block length of T. The blocks are distributed among W workers and multiplied separately, while the results are written to a third matrix shared by all workers. ($N = 2,048; W = 40; T = 16384$)

21. **Quick-Sort** dynamically creates a binary tree of actors where each parent splits the input vector in two and passes each part to a child. The initial input vector is randomly generated with N elements and in each recursive step, the input vector is divided at a partition element. One vector contains elements which are smaller, while the other vector contains elements which are greater or equal. The recursion ends when a vector contains two or less elements. The children return their vectors sorted which then are merged by the parents. ($N = 40,000,000$)

22. **Radix-Sort** arranges sorting actors in a pipeline structure whereby the number of stages is defined by the radix. N randomly chosen values are enqueued to the pipeline. Each sorting actor checks whether the received value has a high bit at the radix position. Values with a high bit are stored in a buffer, while others are forwarded to the next stage. If a stage has received all values, it forwards all buffered values to the next stage, starting with the oldest value. ($N = 400,000$)

24. **Bitonic-Sort** sorts N input items, where N must be a power of 2. The input is arranged into bitonic sequences and then passed through a data independent sorting network consisting of actors. ($N = 8,192$)

25. **Sieve of Eratosthenes** searches for N prime numbers. The sieve is a dynamically arranged pipeline of actors with an incremental number producer at the beginning. Each stage has a buffer for M prime numbers. On receiving a potential prime number, a stage checks whether one of the numbers in the buffer is a divider. When no divider can be found, the number is stored in the buffer and dropped otherwise. If the buffer is full, the potential prime number is passed to the next state. If no next stage exist, a new one is created. ($N = 2000,000$; $M = 10,000$)

27. **Online Facility Location** decides when and where to open a facility in a region based on the cost of opening a facility and the cost of servicing customers. A producer actor feeds a *facility location* actor with N randomly chosen positions of customers. When a density threshold G for customers is reached, the *facility location* actor splits its region in four pieces, spawns a *facility location* actor for each and transfers the respective customers to it. This process is repeated until all customers are placed. ($N = 200,000$; $G = 1,000$)

28. **Trapezoidal Approximation** approximates the integral of the function $f(x) = \frac{1}{x+1} * \sqrt{1 + e^{\sqrt{2x}}} * sin(x^3 - 1)$ between the interval $[L, R]$. The interval is divided into N pieces which are computed by W actors. ($W = 4,000$; $N = 400,000,000$)

6.2.2 The Matrix Search Benchmark

The benchmark *Matrix Search* is designed by us and focuses on a data-intensive task to showcase the benefits of locality-guided scheduling (LGS) over our previous communication locality scheduling (CLS) approach which uses RWS and focuses on CL. Matrix Search can be configured with a number of controllers which coordinate a group of seekers each. All actors are spawned in round robin fashion to balance the workload. A seeker waits for requests from its controller to solve word-finding puzzles. Solving a puzzle requires the seeker to find a sequence of

characters in a local $S * S$ 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. The number of findings is reported back to the controller to signal that the seeker is ready for the next job. A controller sends N puzzle requests (jobs) to each seeker, whereby a job can be assigned in different ways. Jobs can be assigned *continuously* or *block-wise*. In the former case, a seeker finishes its job, reports the result and gets a new job from the controller. In the latter case, all seekers have to be finished before the controller delivers new jobs. The assignment of a job can be further distinguished between *local* and *round robin*. A seeker which reports its findings to its controller switches afterwards to the actor state *Waiting* and remains their until it receives a new job. When receiving a new job, the seeker must be scheduled and enqueued at a job queue of one of the scheduler workers. A local job assignment causes a seeker to be scheduled at the current worker of the controller. This is the default behavior in CAF and designed to increase the cache locality. A round robin assignment schedules seekers accordingly. This avoids piling up all seekers at the same worker and affects the workload balance.

The performance of this benchmark relies on the access characteristics of seekers 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 hereby improve the performance. While the benchmark is an artificial scenario, it is designed to showcase the effect that consideration of locality has on runtime behavior.

6.3 The Data Locality Experience

In this section, we use Matrix Search to compare LGS with our previous scheduling approach CLS. First, we compare their makespan and their data locality. Second, we depict why this approach is naive and give a detailed analysis which reveal performance issues in LGS. Finally, we show how these problems can be solved and summarize the results.

Comparing LGS with CLS

Figure 6.2 shows the real runtime (wall-clock time) of Matrix Search in seconds as a function of the workers of the scheduler and compares multiple versions of LGS and CLS. In this measurement Matrix Search uses one controller and 225 seekers, while jobs are assigned to seekers continuously and locally. Each matrix has a size of 3500 characters in length and width, about 12 MBytes. A seeker searches this matrix 100 times for words with 6 randomly chosen

characters. First we focus on algorithms CLS and LGS which are nearly overlapping for a configuration of up to 12 workers. Thereafter, LGS outperforms CLS with a performance gain of up to 33%. To show that LGS fulfills its design goal we measure the number of local and remote memory accesses, whereby an indicator for good EL is a high number of memory accesses to the local node and a low number to other NUMA-nodes. Consequently, we expect that LGS improves the ratio between local and remote accesses over CLS. Figure 6.3 shows these accesses in absolut numbers as a function of the number of worker threads. Both scheduling algorithms start with a similar ratio of 4% of remote accesses. Then the number of local accesses drops and is replaced by remote accesses. At 32 configured workers CLS leads to more remote accesses than local ones. In contrast, the ratio in LGS fluctuates between 1 and 8% remote accesses and might explain the performance gain.

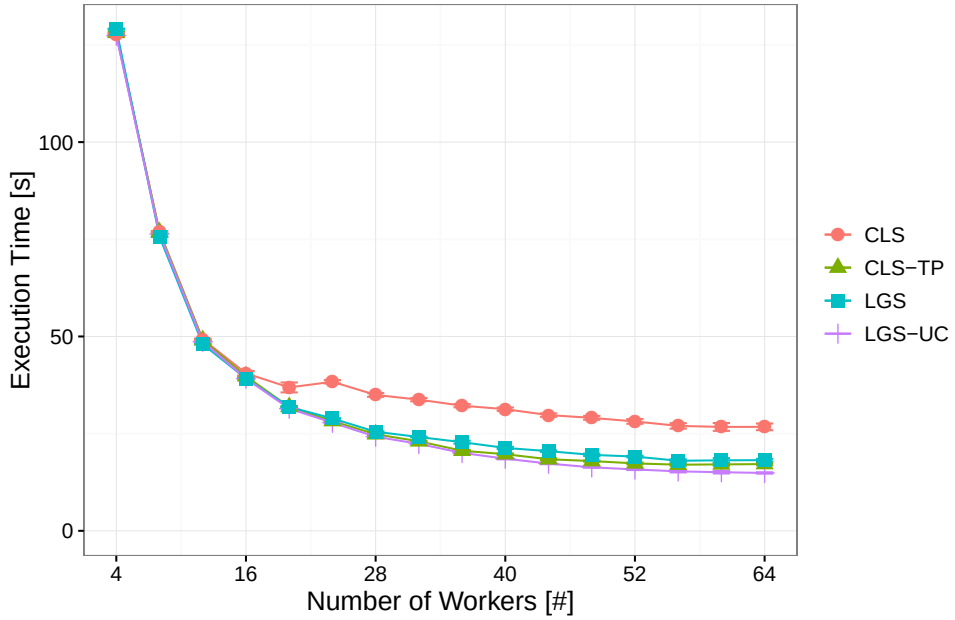


Figure 6.2: Matrix Search – Comparison of the execution time between variants of LGS and the baselines CLS and CLS-TP.

Revealing Performance Problems

Next, in a more detailed analysis we show why the performance gain is not caused by the improved data locality and present that actor pinning can even reduce the performance.

Figure 6.2 also shows a modified version of CLS which we call CLS-TP. CLS-TP behaves exactly like CLS, but each worker of the scheduler is hard-pinned to a specific PU (note that

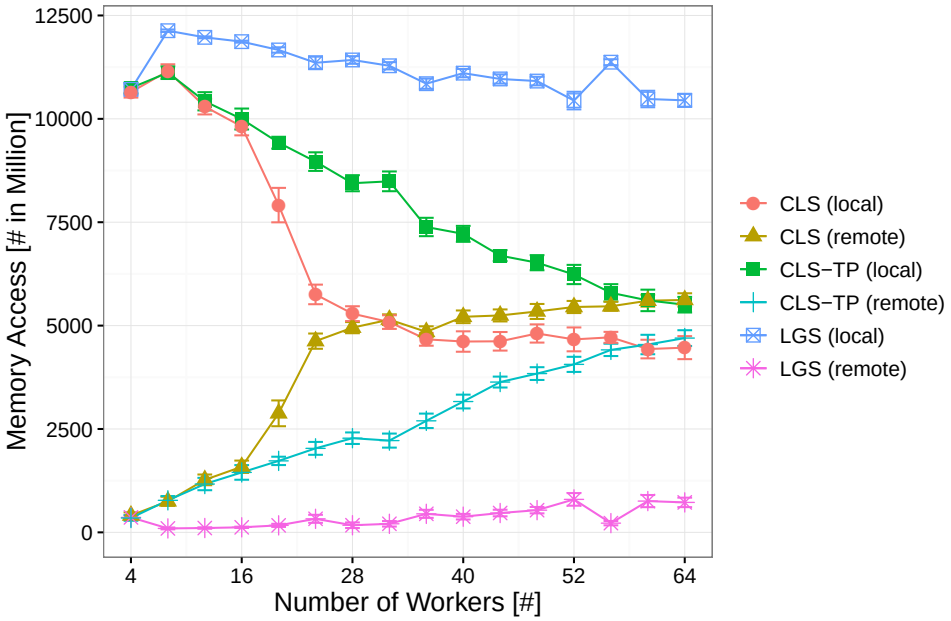


Figure 6.3: Matrix Search – LGS improves the data locality by trading remote with local memory accesses.

the actors are not pinned). This increases the performance of up to 37%. A measurement with Linux Perf explains why pinning of worker threads increases the performance for 64 configured workers. CLS results in about 100,000 migrations of threads between PUs and 4,000,000 page faults in one run. CLS-TP can reduce the migrations to 150 and halves the page faults. Figure 6.4 is a cutout of the same plot to highlight the interesting part. While LGS uses weighted work-stealing and actor pinning, it also requires to pin the worker threads and consequently optimizes the EL of CLS-TP. However, Figure 6.4 shows that LGS is up to 11% slower than CLS-TP. Therefore, only the side effect of thread pinning improves the performance of LGS and the specifically designed data locality optimization reduces it.

Figure 6.3 shows that LGS improves the data locality. After the new insights, one might think this is a side effect of thread pinning as well. However, Figure 6.3 shows the memory accesses for CLS-TP as well and reveals this is not the case. CLS-TP improves the data locality of CLS, but is still much worse than LGS. In detail, the local memory accesses of CLS-TP drop nearly linearly with an increases in the number of workers and are replaced by remote accesses. In comparison, the local accesses of CLS drop notably faster while LGS stays at a constant level with fluctuations.

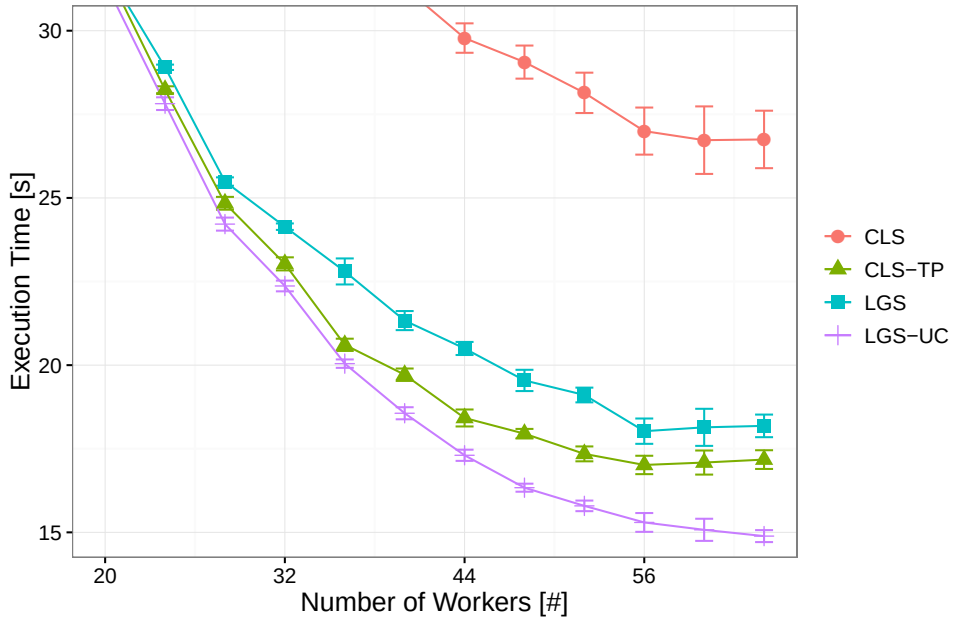


Figure 6.4: Matrix Search – Comparison of the execution time between variants of LGS and the baselines CLS and CLS-TP (zoomed in).

The reason why LGS performs worse than CLS-TP despite better data locality can be explained with Figure 6.5. It examines the activity of the work-stealing scheduler by showing the number of *scheduling event*, *steal attempts* and *successfully steals* in relation to the number of configured workers for CLS, CLS-TP and LGS. Scheduling events are the sum of messages processed by all actors and are independent from the number of scheduling workers. A steal attempt occurs when a worker runs out of jobs and tries to steal a job from another worker, whether or not a job is found. The scheduling events and steal attempts are not related and it is possible to have more attempts than events. Successful steals are the number of stolen jobs and are a subset of attempts and scheduling events. The steal attempts and steals for CLS and CLS-TP have a very similar behavior. While the steals are nearly constant at a low level, the attempts rise linearly from $1,000$ to $30,000$. LGS stands out with a multiple of steals and with a strong fluctuation of attempts. This indicates a load balancing problem which might cause a performance degradation.

Analyzing and Solving the Performance Problem

To identify the cause of the load balancing problem we have to reconstruct the scheduling sequence of the actors. Matrix Search is configured with 1 controller and 225 seekers. All

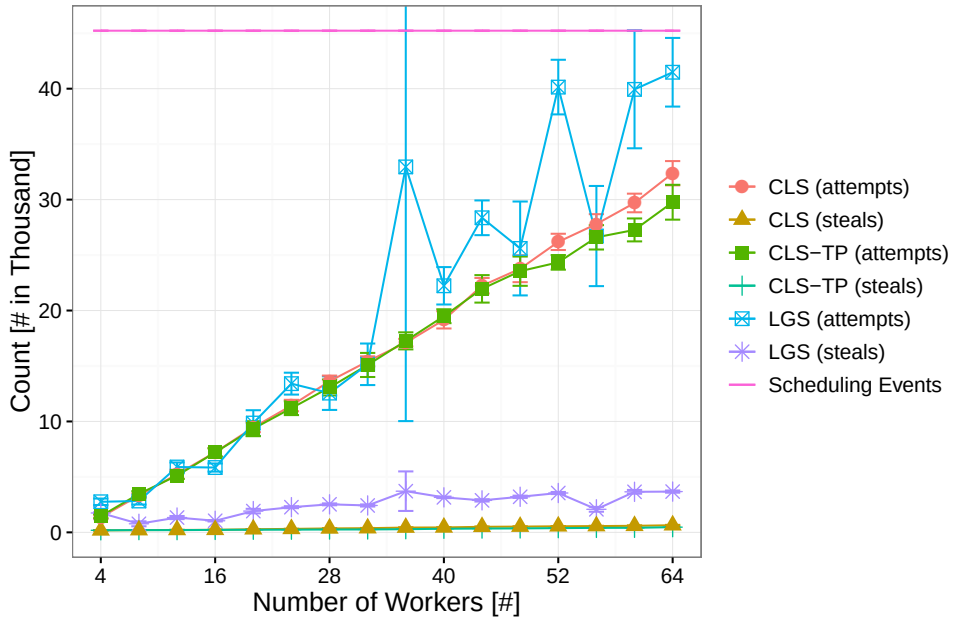


Figure 6.5: Matrix Search – LGS causes fluctuations of steal attempts which might indicate a workload imbalance.

seekers only communicate with the controller which follows the hub and the affinity group pattern identified by Francesquini et al. [63]. In comparison to the controller, seekers have a heavy load. On startup, all actors are evenly distributed in a round robin fashion across the scheduler workers, where they initialize their matrices and process their first job. In average 3.5 seekers are enqueued at each worker. After the first seeker has processed its request, it sends its results back to the controller which has a high change of being idle through its low workload. If the controller is idle, CLS schedules it to be the next job of the current worker. Next, the controller is executed and sends the next job request to the seeker which is again scheduled to be the next actor. As long as the controller is idle, a worker alternates between seeker and controller until the seeker finishes all of its searches. This sequence is very efficient, because the worker is productive the whole time. From the perspective of the controller it jumps from job queue to job queue and performs this sequential CLS pattern for each seeker. When the controller is already enqueued and receives a second message, the pattern is interrupted and causes a workload imbalance. The controller processes both messages one after another and enqueues the respective seekers at the same worker. This is of little consequence as long as each worker as enough work.

LGS behaves differently. After a seeker has processed its request, it sends a message to the controller. If the controller is idle, it is enqueued at the tail of its HPU. Note, it might also be enqueued at the head of one of the direct neighbors of the HPU as shown in Figure 4.3 which can lead to the same sequential CLS pattern. The odds of this pattern to happen is one out of eight, because we have eight NUMA-nodes. Due to the low probability of this pattern we ignore it at this point and focus on the dominant pattern. Enqueuing the controller at the tail can lead to the convoy effect [44]. The convoy effect is a throughput degenerating pattern which occurs when a long running actor like a seeker blocks other actors from execution. The execution of the controller is critical for the performance of Matrix Search. If it is stalled in a job queue behind seekers, workers might run out of jobs. This can cause fluctuations of steal attempt and an increased number of steals as shown in Figure 6.5.

To prove our hypothesis of the convoy effect we modify LGS to prevent this effect and call it LGS-UC (Unpinned Controller). While LGS pins all actors to its HPU, LGS-UC allows to unpin specific actors manually. We unpinned the controller which then behaves similar to a CLS scheduled controller. This avoids the convoy effect, allows the sequential CLS pattern and profits from the improved data locality of LGS. Figure 6.2 and 6.4 compare the execution times of LGS-UC with our other scheduling algorithms. They show that LGS-UC reduces the execution time of CLS-TP by up to 13% and it consequently confirms the hypothesis.

In summary, we are able to improve the performance of a custom tailored benchmark by up to 44%. It has the drawback that it requires hints of the application developer of which actors have to be pinned and which not and providing such hints requires deep knowledge about the scheduling in CAF. However, without hints of the application developer it is still possible to increase the performance with CLS-TP by up to 37% and with LGS by up to 33% over CLS. In the following subsections we describe further effects seen with LGS and Matrix Search.

6.4 Matrix Search Measurements

In this section we present further insights from our experiments with the Matrix Search benchmark. First, we show additional baselines LGS can be compared with. We study two variants of LGS which might improve the performance. Finally, we show a use case where LGS has a clear advantage over CLS.

6.4.1 The Interleaved Memory Access Mode

To examine the performance gain of LGS we used CLS and CLS-TP as baselines, because they require no or little effort to use in a productive environment. Other baselines we can compare

LGS with are the memory interleaved versions of CLS and CLS-TP. Our test system is per default configured with the memory policy *first touch* which allocates memory on the NUMA-node the worker thread is currently running on. In contrast, the *interleaved* policy allocates memory evenly from all NUMA-nodes and is designed to avoid worst case memory accesses. We configured our test system to use this interleaved mode and measured the execution time of Matrix Search with CLS and CLS-TP.

Figure 6.6 shows the results. *CLS (interleaved)* is up to 30% faster than CLS and nearly reaches the execution time of LGS. While CLS-TP is already 38% faster than CLS, the interleaved version even exceeds the performance of LGS-UC with all equipped workers and is up to 47% faster than CLS. The performance of CLS-TP (interleaved) is about 3% lower than LGS-UC for the range of 4 to 52 workers. Afterward, the performance of CLS-TP (interleaved) surpasses LGS-UC. It is unexpected that CLS-TP in the interleaved mode outperforms our specifically tailored scheduling algorithm LGS-UC and requires further analysis.

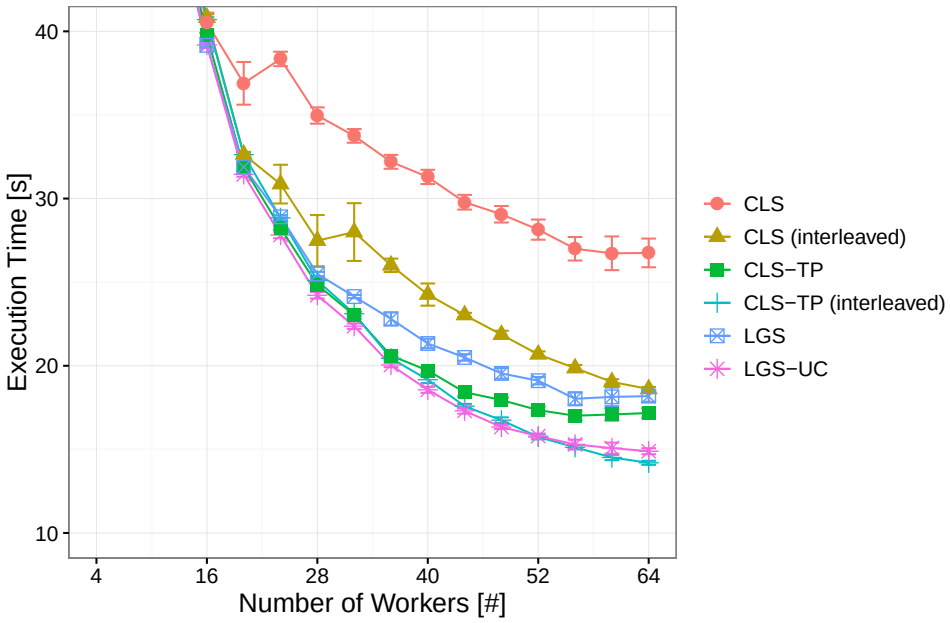


Figure 6.6: Matrix Search – CLS-TP (interleaved) outperforms our custom tailored scheduling algorithm LGS-UC (zoomed in).

6.4.2 LGS Variants

Optimizing the scheduling in CAF is a complex challenge. Many interacting parameters can be adjusted, little changes can have a big impact and can cause contrasting results in different

benchmarks. These characteristics make it hard to predict which changes have the best result. In this section, we present further LGS variants which seem promising based on our previous knowledge, but are dead ends. We focus on the benchmark Matrix Search and a detailed analysis.

In the previous section we showed that LGS is slower than CLS-TP due to by the convoy effect. LGS-UC avoids this effect by manually unpinning the controllers. Hence, the convoy effect only occurs in the Breadth First Search (BFS) scheduling strategy, using Depth First Search (DFS) would also solve the problem. The LGS version which substitutes BFS with a DFS strategy is called LGS-DFS. It is implemented by enqueuing idle actors at their HPU at the head instead of at the tail. Originally, we designed LGS to enqueue actors at the tail of the job queues to avoid interfering with local cache optimizations. LGS-DFS has the advantage over LGS-UC that no effort from an application developer is necessary to avoid the convoy effect. On the downside, enqueuing jobs at the head from a different worker might trash their cache and consequently cause a degradation in performance. LGS-UC preserves the efficient CLS sequential pattern while LGS-DFS cannot do so, because the controller is still pinned to a specific HPU. A further drawback of this approach is that enqueuing jobs at the head requires to acquire two locks instead of one to synchronize access to the job queue. We expect LGS-DFS to be slower than LGS-UC due to the interrupt of the CLS sequential pattern and the impact of the cache trashing is presumably low. The working set of a seeker exceeds the CPU caches by multiple times and each execution will trash the cache anyway. Trashing of cached messages is also expected to be insignificant, because they have a low memory footprint, are send in a low frequency and are transmitted to pinned actors which mostly do not share a cache with the controller. The expense of the additional synchronization is unclear.

Figure 6.7 compares the execution time of LGS-DFS with LGS and LGS-UC. Despite avoiding the convoy effect LGS-DFS is up to 19% slower than LGS and shows that switching from BFS to DFS does not solve our performance problems.

Figure 6.7 shows another variation of LGS called LGS-CO (LGS with Cache Optimizations). The direct neighbors group g_0 of LGS contains all PUs of a NUMA-node while we restricted the direct neighbors of LGS-CO to PUs sharing the same L2-cache. It reduces the size of g_0 for LGS-CO from eight PUs to two and affects weighted work-stealing. The actor pinning behavior is not changed. The cache locality optimization has no effect on Matrix Search and shows that Matrix Search is very insensitive to cache optimizations.

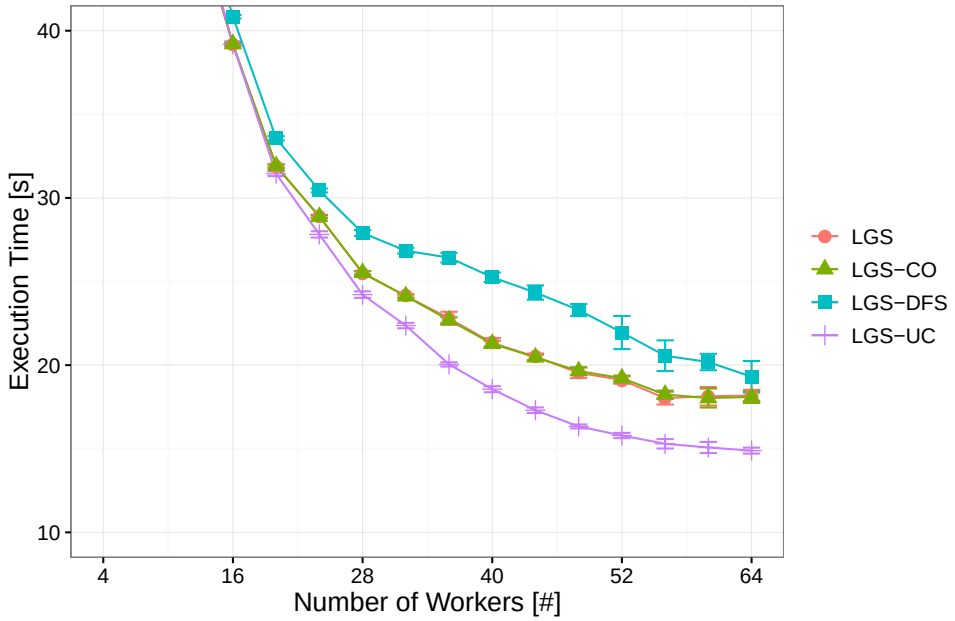


Figure 6.7: Matrix Search – LGS-UC outperforms our other experimental schedulers (zoomed in).

6.4.3 Scalability of Job Assignment Patterns

LGS changes the scheduling behavior of CLS. While this can cause a degradation in performance, it can also improve the scalability for specific communication patterns like the block-wise assignment pattern. In our previous Matrix Search measurements we assigned jobs continuously such that when an actor finish its work it informs the controller and immediately receives the next job request. In a block-wise assignment an actor only receives a new job once all other actors have also finished their jobs. A controller which uses this pattern in conjunction with local assignment of jobs can cause scalability problems. The actor of a local assigned task is enqueued at the job queue of the current worker of the controller. While block-wise assignment is an application specific behavior, locally assigning of jobs is the default method in CLS. Figure 6.8 shows the execution time of the Matrix Search benchmark as a function of the number of workers. Matrix Search was configured with one controller, 225 seekers, a matrix with 3500 bytes in length and width and all jobs are assigned block-wise. The graph *CLS (local)* and *CLS-TP (local)* assign their jobs locally which leads to the peek performance at 28 workers. Adding more workers is counter productive and lengthens the makespan significantly. This can be explained by two reasons. (1) The block-wise assignment of jobs interrupts the CLS sequential pattern and (2) a local combined with a block-wise assignment piles up all

jobs at the same worker at once. The former increases the idle time of workers and the latter leads to the worst case scenario of an imbalanced workload. Work-stealing handles such a workload hotspot very ineffectively, because the odds that a thief find this hotspot declines with the number of victims. After a number of unsuccessful attempts a thief even starts short intermediate sleeps to reduce the CPU load. CLS-TP has the same scalability issue, but an overall increased performance due to a reduced number of thread migrations between PUs and an increased data locality. This scalability problem can be solved by manually forcing CAF to spread the jobs evenly over the workers. The graph *CLS (round robin)* shows this and indicates a much better performance. LGS inherently solves this with actor pinning and requires no effort from the application developer. When actors are evenly pinned over all scheduler workers a block-wise assignment cannot cause a workload hotspot.

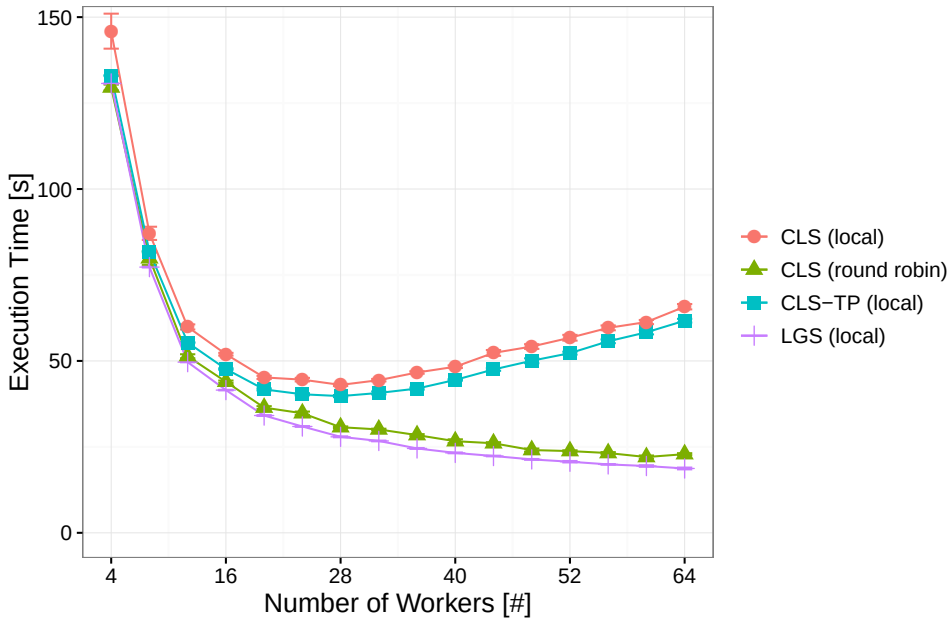


Figure 6.8: Matrix Search – LGS can prevent workload hotspots and related scalability problems.

6.5 Savina Measurements

In this section we study LGS with the Savina benchmarks. First, we summarize the performance and data locality characteristics of LGS. We compare LGS and CLS-TP with CLS and examine the advantage of LGS-CO and the interleaved mode over LGS. Subsequently, we analyze several benchmarks in detail and reveal multiple scheduling issues.

6.5.1 The Performance and Data Locality Summary

Comparing LGS with CLS

To summarize the overall performance of LGS compared to CLS in the Savina benchmarks we performed measurements for both schedulers using 64 workers as shown in Figure 6.9. 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 *25-Apsp* benchmark. Note that the graph only shows an overall trend and does neither provide information about error distribution nor scalability.

6 out of 23 Savina benchmarks perform better with LGS than with CLS. The benchmarks *25-Apsp*, *30-Bitonicsort* and *09-Concsll* show excellent results with a reduced execution time of over 30%. In contrast, *10-Bndbuffer* is nearly 9 times slower and LGS more than doubles the execution time of *14-Logmap*, *20-Facloc* and *07-Big*. To understand the impact of LGS in each benchmark a detailed analysis is required, because the performance gain or loss can have diverse causes. This includes the convoy effect as demonstrated with the Matrix Search benchmark in Section 6.2.2, modifications of the balance of the workload or other scheduling issues.

Next, we want to show how LGS affects the data locality of the Savina benchmarks. Figure 6.10 and Figure 6.11 compare the number of local and remote memory accesses between CLS and LGS. The baselines in both figures signify the mean of the number memory accesses of CLS over 10 measurements. A lower percentage than 100 shows that LGS reduces the number of accesses while a higher percentage shows an increase. Of the 23 Savina benchmarks, LGS reduces the number of remote accesses in 16 cases and increases local accesses in 15 cases. This suggests that LGS increases the data locality in about 70% of the cases. However, only 26% of the benchmarks have an improved performance which indicates that in most benchmarks the data locality optimizations play a minor role or are covered by scheduling issues.

A closer look at Figures 6.9, 6.10 and 6.11 reveals multiple interesting points. First, the results are diverse and seem contradicting. For example, the benchmark *03-Fjthrput* is 17% slower with LGS compared to CLS despite less local and remote accesses. In contrast, *25-Apsp* is 36% faster and has less local and remote accesses. Second, some benchmarks have in total much more or much less memory accesses without changing the problem complexity. For example, benchmark *03-Fjthrput* has 27% less remote accesses as well as 16% less local accesses. This contradicts with our experience from our previous experiments. We show in Section 6.3 that LGS reduces slow remote memory accesses in the benchmark Matrix Search and trades it for fast local memory accesses. We conjecture that reducing the number of remote memory

accesses can improve the number of cache hits on the local NUMA-node and consequently reduce the overall accesses to the main memory. In consequence, an increased number of remote accesses might increase the cache miss rate. For example, *08-Concdict* has 40% more remote accesses and 55% more local access. A workload imbalance can be another reason which increases the overall memory accesses due to many steal attempts. In the Sections 6.5.4, 6.5.3, 6.5.2 we examine several outstanding benchmarks in detail.

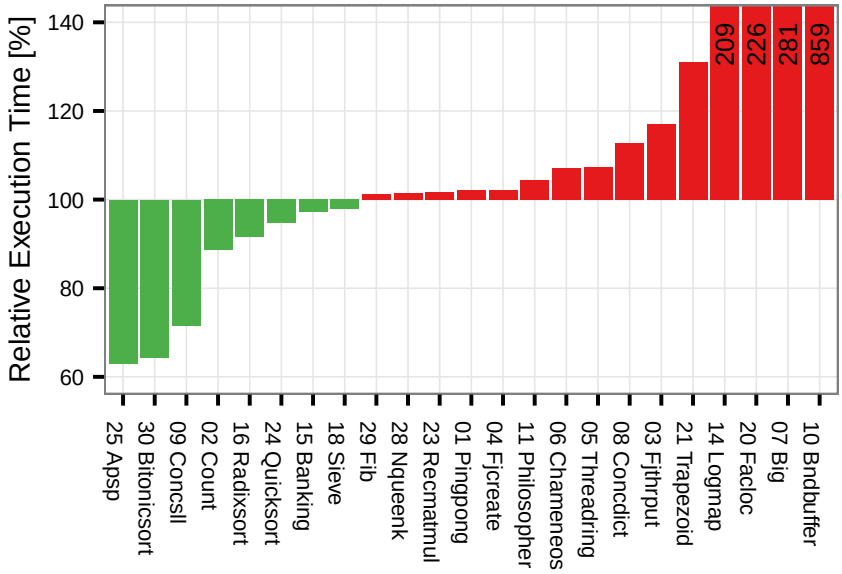


Figure 6.9: Savina benchmarks – Comparison of the makespan between LGS and CLS (baseline).

Comparing CLS-TP with CLS

CLS-TP is a variant of CLS where the scheduler workers are hard pinned to specific PUs. This prevents the scheduler from migrating threads between PUs. As presented in Section 6.3, CLS-TP improves the performance of the benchmark Matrix Search by up to 37% in comparison to CLS. In this section, we want to summarize the impact of CLS-TP on the performance of the Savina benchmarks. Figure 6.12 shows the relative execution time with CLS as a baseline (100%), measured with 64 scheduler workers. Only 7 out of 23 benchmarks profit from CLS-TP while the other benchmarks have an increased execution time. The performance loss or gain is in the range of $\pm 10\%$ for all benchmarks except for *03-Fjthrput* and *14-Logmap*. They stand out with a performance loss of 33% and 52%, respectively. Further measurements reveal that 20 of the benchmarks have less remote memory access and 16 benchmarks less local memory

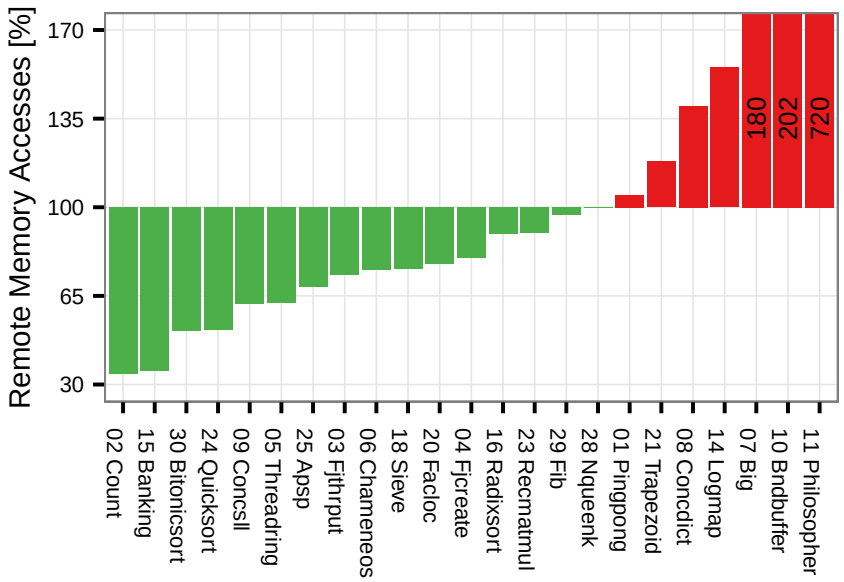


Figure 6.10: Savina benchmarks – Comparison of remote memory accesses between LGS and CLS (baseline).

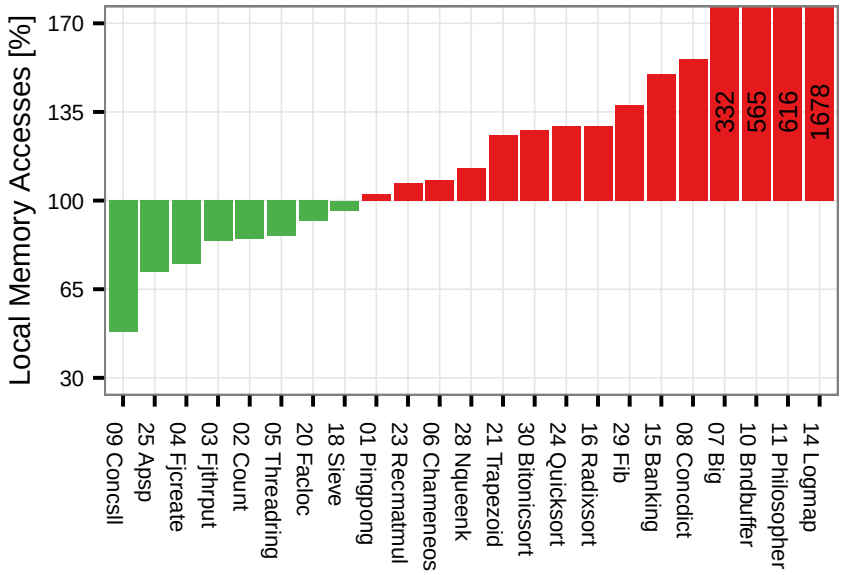


Figure 6.11: Savina benchmarks - Comparison of local memory accesses between LGS and CLS (baseline).

accesses. Despite the increase of the data locality in most cases CLS-TP performs worse than CLS. This indicates that other factors outweigh the increased data locality.

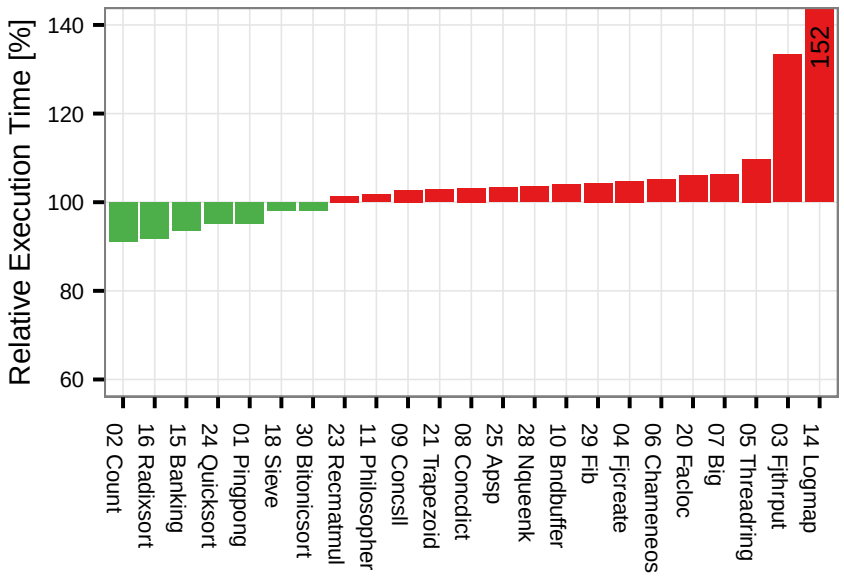


Figure 6.12: Savina benchmarks – Comparison of the makespan between CLS-TP and CLS (baseline).

Comparing LGS-CO with LGS

LGS-CO is a modification of LGS designed to improve the cache locality. As described in Section 6.4.2, LGS-CO improves the cache locality by stealing jobs from workers sharing the same L2 cache first before expanding the range to a whole NUMA-node. Figure 6.13 shows the performance of the Savina benchmarks with LGS-CO. It shows the relative execution time of of LGS-CO with LGS as a baseline. The performance of 14 out of the 23 Savina benchmarks has changed less than 3%. The benchmarks *10-Bndbuffer* and *20-Facloc* stand out with a performance gain of 36% and 67% performance loss, respectively. The performance gain of 36% seems like a good result, but the benchmark is still more than four times slower than with CLS in use. In summary, LGS-CO rarely has advantage over LGS.

The Interleaved Mode

The interleaved mode is a memory allocation pattern which allocates memory evenly from all NUMA-nodes. In Section 6.4.1 we compared different baselines and found that CLS-TP used

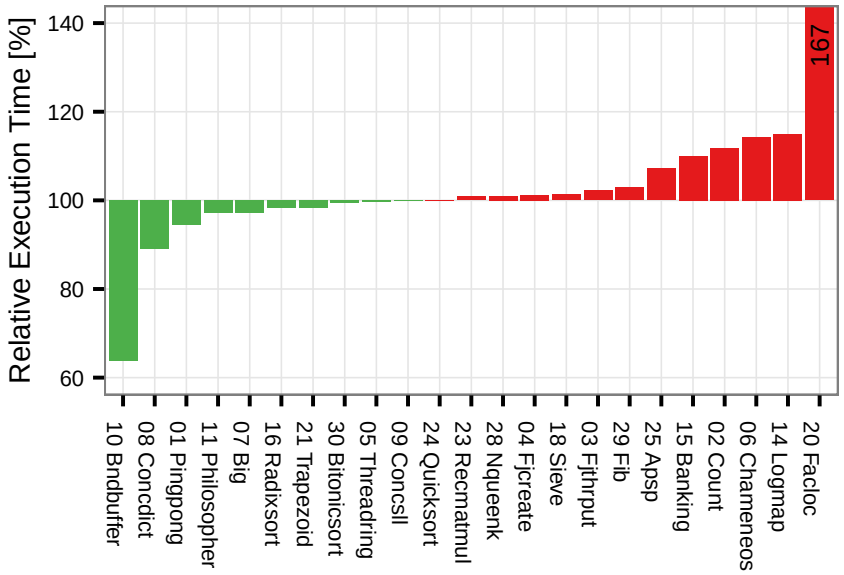


Figure 6.13: Savina benchmarks – Comparison of the makespan between LGS-CO and CLS (baseline).

in the interleaved mode can outperform LGS. In this section, we examine whether *CLS-TP (interleaved)* and *CLS (interleaved)* outperform LGS in the Savina benchmark suite. Figure 6.14 and 6.15 show the relative execution time for *CLS (interleaved)* and *CLS-TP (interleaved)* with CLS as a baseline. *CLS (interleaved)* increases the performance of about half of the Savina benchmarks but also decrease the other half. The performance of most of the benchmarks varies between -4% and +4% while the benchmarks *30-Bitonicsort* and *24-Quicksort* stand out with -15% and +26%, respectively. *CLS-TP (interleaved)* only improves 6 out of 23 benchmarks which is one benchmark less then with CLS-TP as shown in Figure 6.12. Apart from that, *CLS-TP (interleaved)* has a similar performance characteristics to CLS-TP. In summary, *CLS-TP (interleaved)* and CLS-TP have very similar performance characteristics. *CLS (interleaved)* and *CLS-TP (interleaved)* are not a general substitute for CLS and provide only an improvement over CLS, CLS-TP or LGS in rare cases.

6.5.2 Pushing Actors to Sleeping Workers

The Savina benchmark *01-Ping-Pong* heavily favors communication locality. As the name suggests, it deploys two actors that exchange a defined number of messages, in our case $2 * 10^6$.

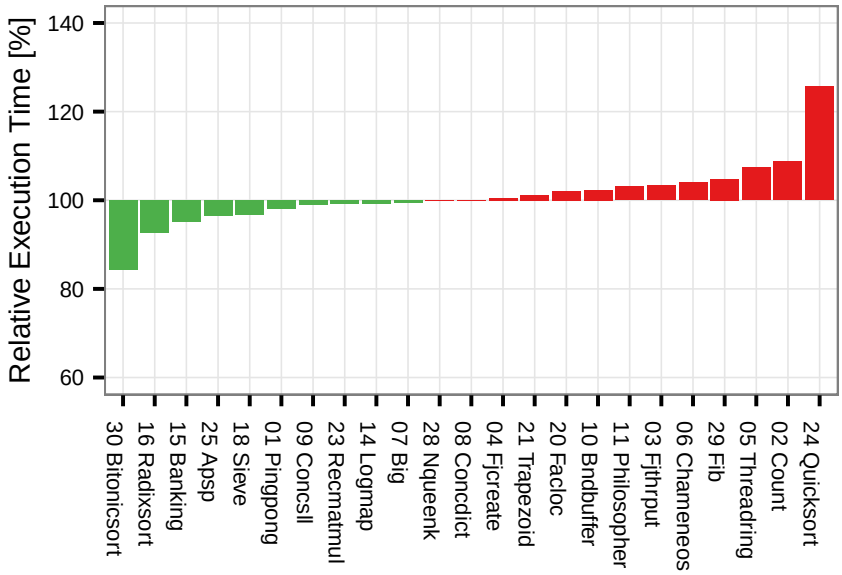


Figure 6.14: Savina benchmarks – Comparison of the makespan between *CLS (interleaved)* and CLS.

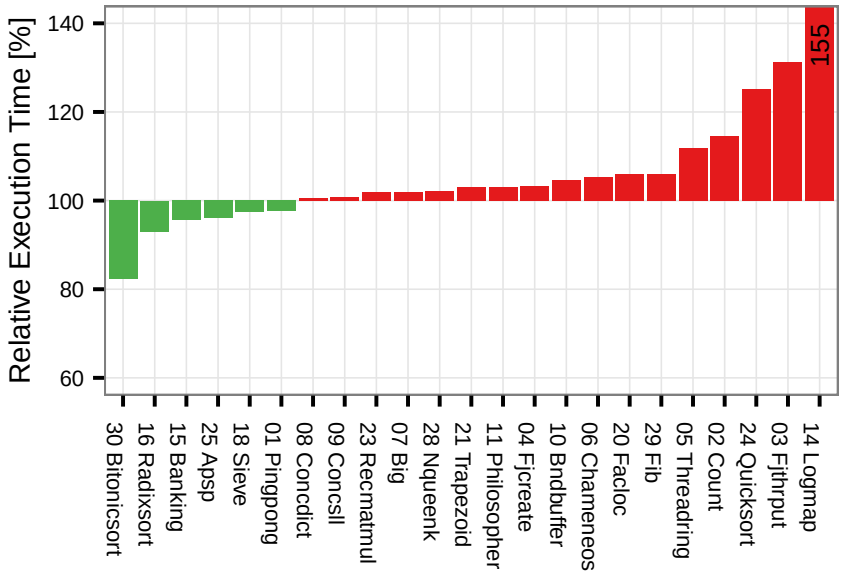


Figure 6.15: Savina benchmarks – Comparison of the makespan between *CLS-TP (interleaved)* and CLS.

Figure 6.16 shows the runtime as a function of the number of workers for the benchmark. Both CAF deployments, CLS and LGS, outperform the JVM-based frameworks Akka and Habanero. 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 a job is pushed to a sleeping worker. This behavior was discussed in Section 4.3 alongside a mitigation strategy that is implemented for the LGS measurements.

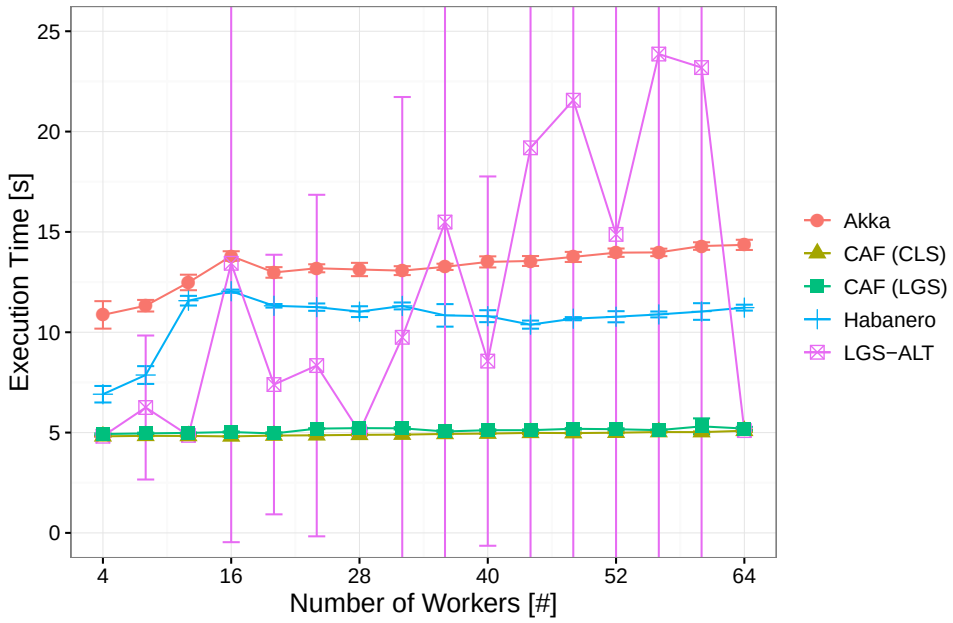


Figure 6.16: Savina 01-PingPong – Two actors repeatedly exchange messages in a ping-pong style communication pattern.

Part of the mitigation strategy prevents actors from adopting a distant node as the HPU if they are stolen before their first execution. This allows LGS to avoid intermediate sleeps of involved workers as well as communication between distant NUMA-nodes. To show that the behavior of LGS-Alt is not solely explained by the communication distance we performed a measurement where we prevented workers from sleeping. Under those condition, LGS-Alt still performs 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.

6.5.3 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* (*08-Concdict*) and *Concurrent Sorted Linked-List* (*09-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. Concdict uses a dictionary with a read/write complexity of $O(1)$ while Concsll uses a sorted linked list with a complexity of $O(N)$. Note, the benchmarks cannot be compared directly. Due to their different complexity, Concdict is performed with 100 actors, each sending 50 thousand messages, while Concsll is performed with 20 actors, each sending 8 thousand messages.

Figure 6.17 and 6.18 depict the makespan for Concsll and Concdict in seconds as a function of the number of scheduler workers. We run the benchmarks with the parameters as described in Section 6.2.1. For Concsll, 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 outperforms the other CAF scheduler by up to 32.5%. For Concdict, both CAF scheduling strategies perform well 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 Concdict, 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 Concdict, because it traverses the sorted linked list on every read and write access. In contrast, the dictionary of Concdict has a constant access time and a low access complexity which reduces the importance of EL and shifts the focus towards CL. Figure 6.19 and 6.20 show local and remote memory accesses of Concsll and Concdict. For Concsll, LGS improves the execution locality by reducing the number of remote accesses and the cache locality by reducing the number of local memory accesses. For Condict, it causes an increase of the overall memory accesses.

This knowledge can be exploited to improve the performance of Concdict with a modification of LGS. While non data actors profit from an high CL, the data actor might still profit from an high EL. As a test, we combined these contrary goals by unpinning all actors, but the data actor and call this scheduling strategy LGS-UW (Unpinned Worker). As shown in Figure 6.18, this approach slightly improves the performance by up to 4%.

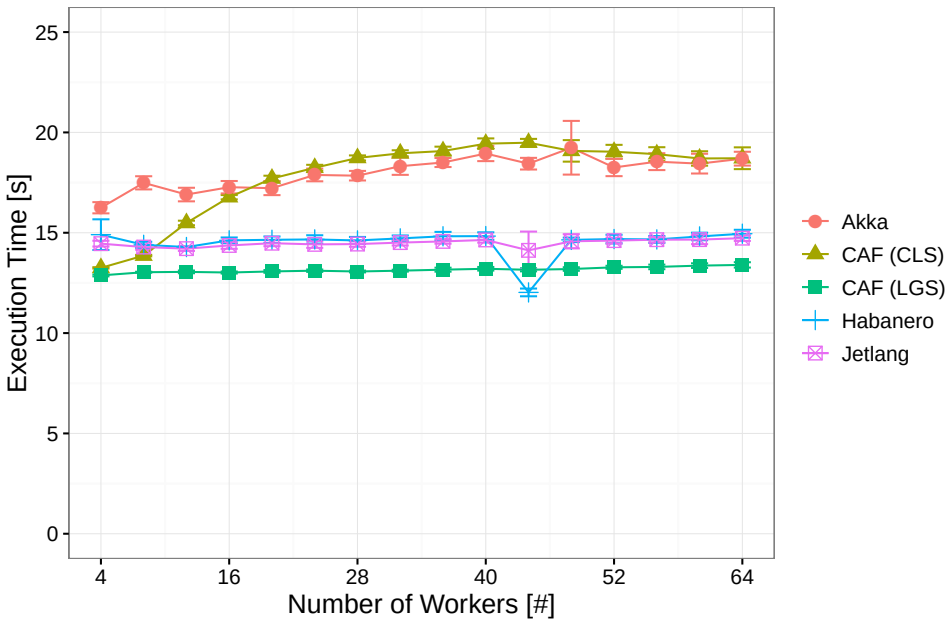


Figure 6.17: Savina 09-Concsll – Actors synchronize accesses into a linked list through a central coordinator.

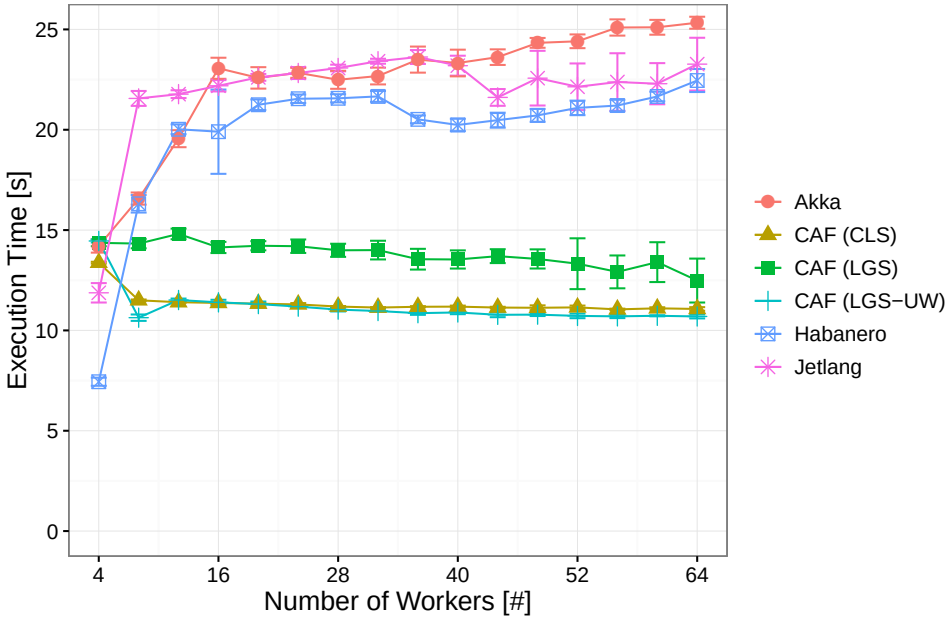


Figure 6.18: Savina 08-Concdict – Actors synchronize accesses into a dictionary through a central coordinator.

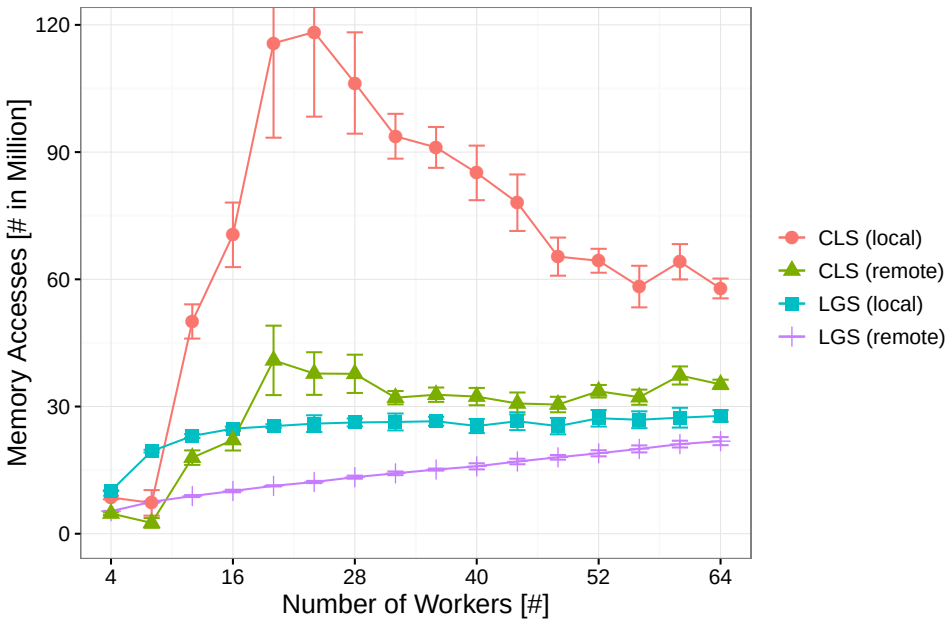


Figure 6.19: Savina 09-Concsll – LGS reduces the overall memory accesses by improving the execution locality as well as the cache locality.

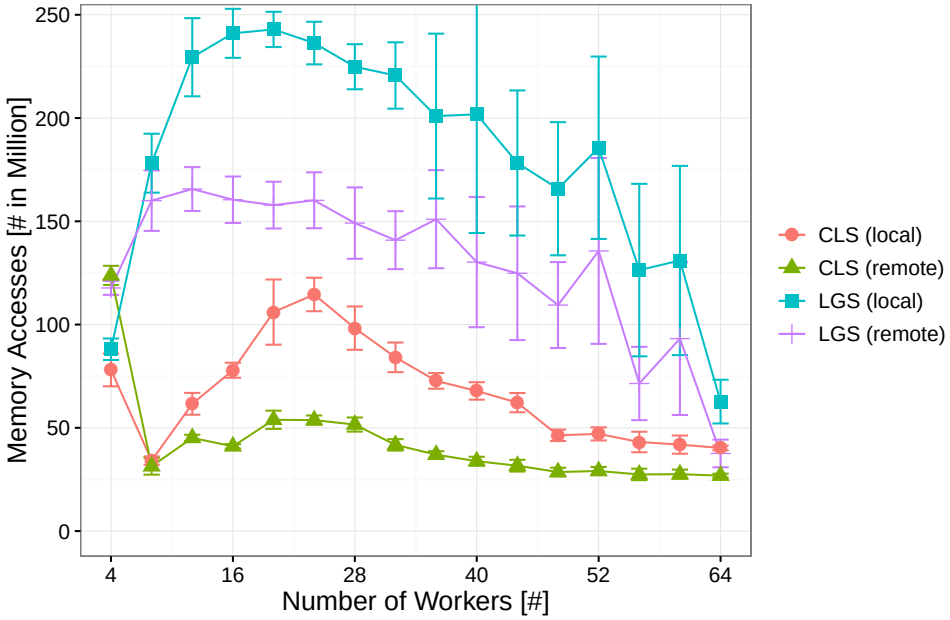


Figure 6.20: Savina 08-Concdict – LGS reduces the communication locality and causes an increase of the overall memory accesses.

6.5.4 A Cause for a High Number of Steal Attempts

For some benchmarks, we observed that LGS causes a very high number of steal attempts compared to CLS. This includes the benchmarks *10-Bndbuffer*, *14-Logmap*, *07-Big*, and *11-Philosopher*. In the following, we investigate the benchmark *11-Philosopher* and show a scheduling pattern which leads to a high number of steal attempts.

The Benchmark *11-Philosopher* stands out with about 50 times more steal attempts with LGS compared to CLS. Furthermore, it has about 7 times more remote memory accesses and 6 times more local accesses. However, the performance is only 4% worse at 64 cores. The Dinning Philosopher is a classical synchronization problem. Multiple resources are shared between workers and each worker requires multiple resources exclusively to do a specific task. In this benchmark, the problem is solved by an arbitrator actor which manages all resources while philosophers request them in a polling manner. If both required resources are free, the arbitrator assigns them to the requester and denies the request otherwise. Figure 6.21 shows the number of scheduling events, steals and steal attempts for LGS and CLS as a function of the number of scheduler workers. A detailed description of these events is given in Section 6.3.

In addition to the high number of steal attempts, the non constant number of scheduling events is conspicuous. LGS has on average 7% less scheduling events than CLS. We guess this is due to the different scheduling sequences of actors in CLS and LGS. While the former uses a depth first search (DFS) strategy, the latter uses mix of DFS and breadth first search (BFS). This changes the order of actors in the job queues and the odds for accepted requests of the philosophers. *11-Philosopher* is the only benchmark with a non constant number of scheduling events.

The high number of steal attempts has the following cause. On startup, 80 philosopher actors are spawned, evenly distributed over all scheduler workers and pinned. The arbitrator is a bottleneck in this benchmark and most philosophers are idle and wait its response. Consequently, most scheduler workers are idle as well and try to steal work from other workers which increases the number of steal attempts. After a preconfigured number of steal attempts, a worker starts intermediate sleep to reduce the contention on the job queues and to reduce the CPU load. The sleep intervals extend over time to reduce the CPU load further. After a successful steal or execution of an actor the sleep duration is reset. LGS pushes idle philosophers to their HPU. The next time the HPU wakes up, it checks its own job queue and execute the philosopher which resets the sleep interval. In CLS, the sleep interval is only reset after a successful steal. Therefore, LGS resets the sleep interval of workers more often than CLS and causes a higher number of steal attempts. This increases the number of local and remote memory accesses, increases the contention at job queues and can increase the makespan.

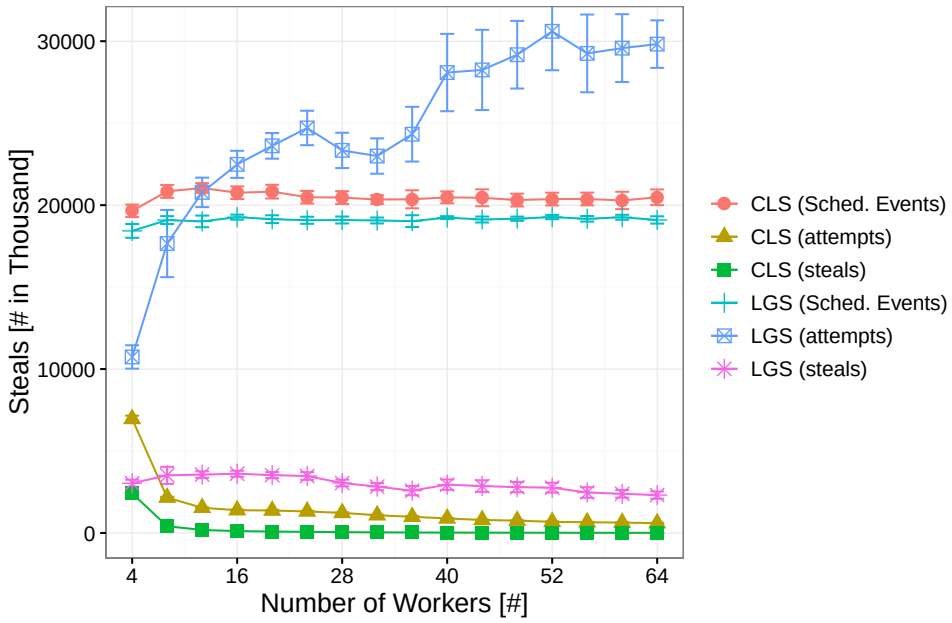


Figure 6.21: Savina 11-Philosopher – A central arbitrator manages shared resources.

7 Conclusion and Outlook

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

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 focus of the default CAF scheduler (CLS) from communication locality (CL) towards execution locality (EL). While CL influences the performance of inter-actor message exchange, EL affects the execution time of actors. These types of data locality may conflict, hence LGS aims at a trade-off between CL and EL.

LGS combines a weighted work-stealing approach with soft actor pinning. The former is a specialization of the random work-stealing scheduling algorithm and balances the workload by preferably picking victims from memory vicinity. The latter schedules actors close to their initial worker for facilitating fast access of their state. Both approaches require no additional effort from application developers and have little runtime overhead.

We performed extensive measurements of LGS to evaluate the effects on data locality and performance. We designed a data-intensive Matrix Search benchmark and translated the JVM-based actor benchmark suite Savina to C++. The former is designed to showcase the benefits of LGS, while the latter allows to explore the behavior of LGS in a wide range of diverse scenarios.

First experiments with LGS and Matrix Search showed significant performance and data locality improvements. However, a detailed study reveals that these improvements are caused by a side effect and actor pinning can even reduce the performance. The performance degradation is caused by scheduling problems like the convoy effect. With a custom tailored version of LGS which allows to unpin specific actors manually, we can avoid this scheduling effect and improve the performance of Matrix Search by up to 44% compared to CLS.

LGS applied to the Savina suite improves the data locality of 16 out of 23 benchmarks. However, only 8 benchmarks have an improved performance which indicates that various benchmarks suffer from scheduling problems similar to Matrix Search. A detailed investigation of several Savina benchmarks confirmed this.

In summary, LGS can improve data locality and performance, but it is unsuitable as the default scheduler in CAF. Due to several scheduling problems we identified, it requires manual adjustments which is only profitable in special cases.

We also experimented with Linux standard tools and configurations which allows thread pinning and switching between different memory access modes and showed the impact on performance and data locality. While we significantly improved the performance of Matrix Search with such methods, Savina benchmarks only profited in rare cases.

In our future work we will extend CAF with a memory management layer. This enables actors to allocate memory from specific NUMA-nodes and can avoid memory scattering across nodes and consequently improves the data locality. A CAF specific memory management layer also reduces the cost of memory allocations. In contrast to a generic memory management, CAF has deep knowledge of its memory specific requirements like the size of actors and messages which allows to improve prefetching and recycling of memory.

Bibliography

[1] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proc. of the 3rd IJCAI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[2] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments," in *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2013, pp. 87–96.

[3] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting Actor Programming in C++," *Computer Languages, Systems & Structures*, vol. 45, pp. 105–131, April 2016. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2016.01.002

[4] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[5] K. L. Johnson, "The Impact of Communication Locality on Large-scale Multiprocessor Performance," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 392–402, 1992.

[6] M. Pericas, A. Cristal, R. Gonzalez, D. A. Jimenez, and M. Valero, "A decoupled KILO-instruction processor," in *The 12th International Symposium on High-Performance Computer Architecture, 2006.* Berlin, Heidelberg: Springer-Verlag, Feb 2006, pp. 53–64.

[7] S. Imam and V. Sarkar, "Savina – An Actor Benchmark Suite," in *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2014, pp. 67–80.

[8] J. De Koster, T. Van Cutsem, and W. De Meuter, "43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties," in *Proc. of the 6th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '16), Workshop AGERE!* New York, NY, USA: ACM, 2016.

[9] J. Armstrong, "A History of Erlang," in *Proc. of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*. New York, NY, USA: ACM, 2007, pp. 6–1–6–26.

[10] G. Agha, *Actors: A Model of Concurrent Computation In Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[11] B. C. Smith and C. Hewitt, "A Plasma Primer (draft)," 1975.

[12] Typesafe Inc., "Akka Framework," http://akka.io, August 2017.

[13] S. Imam and V. Sarkar, "Habanero-Java Library: A Java 8 Framework for Multicore Programming," in *PPPJ*. ACM, 2014, pp. 75–86.

[14] M. Rettig, "Jetlang," https://github.com/jetlang, accessed: 2017-07-18.

[15] Scalaz Developers, "Scalaz: Scala Library for Functional Programming," https://github.com/scalaz, accessed: 2017-07-18.

[16] Theron Developers, "Theron C++ Concurrency Library," http://www.theron-library.com, accessed: 2017-11-09.

[17] L. V. Kale and S. Krishnan, "Charm++: Parallel programming with message-driven objects," *Parallel Programming using C++*, pp. 175–213, 1996.

[18] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil, "Deny Capabilities for Safe, Fast Actors," in *Proc. of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ser. AGERE! 2015. New York, NY, USA: ACM, 2015, pp. 1–12.

[19] J. Armstrong, "Erlang - A Survey of the Language and its Industrial Applications," in *Proc. of the symposium on industrial applications of Prolog (INAP96)*. Hino, October 1996, pp. 16–18.

[20] D. Charousset, R. Hiesgen, and T. C. Schmidt, "CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications," in *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2014, pp. 15–28.

[21] M. Triebe, D. Charousset, R. Hiesgen, and T. C. Schmidt, "Das C++ Actor Framework im Leistungsvergleich," in *Report 302, 8. GI/ITG Workshop Leistungs-, Zuverlässigkeits-*

und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen (MMBnet15). Hamburg, Germany: Universität Hamburg, Dept. Informatik, October 2015, pp. 83–89.

[22] R. Hiesgen, D. Charousset, and T. C. Schmidt, "Manyfold Actors: Extending the C++ Actor Framework to Heterogeneous Many-Core Machines using OpenCL," in *Proc. of the 6th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '15), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2015, pp. 45–56.

[23] R. Hiesgen, D. Charousset, T. C. Schmidt, and M. Wählisch, "Programming Actors for the Internet of Things," *Ercim News*, vol. 101, pp. 25–26, April 2015. [Online]. Available: http://ercim-news.ercim.eu/en101/special/programming-actors-for-the-internet-of-things

[24] S. Wölke, R. Hiesgen, D. Charousset, and T. C. Schmidt, "Locality-Guided Scheduling in CAF," in *Proc. of the 8th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '17), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2017, pp. 11–20.

[25] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 180–186.

[26] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*. Wiley-Interscience, 2005.

[27] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*, 1st ed. Chapman & Hall/CRC, 2015.

[28] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[29] Advanced Micro Devices Inc., "IOS and Kernel Developers Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors," http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf, accessed: 26/10/2017.

[30] P. J. Denning, "The Locality Principle," *Commun. ACM*, vol. 48, no. 7, pp. 19–24, 2005.

[31] J. B. Chen and B. D. D. Leupen, "Improving Instruction Locality with Just-in-time Code Layout," in *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, ser. NT'97. USENIX Association, 1997.

[32] H. Gao, Y. Ma, M. Dimitrov, and H. Zhou, "Address-Branch Correlation: A Novel Locality for Long-Latency Hard-to-Predict Branches," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, Salt Lake City, UT, USA, 2008, pp. 74–85.

[33] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[34] P. J. Denning, "Working Sets Past and Present," *IEEE Trans. Softw. Eng.*, vol. 6, no. 1, pp. 64–84, 1980.

[35] T. David, R. Guerraoui, and V. Trigonakis, "Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 33–48.

[36] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview," *Queue*, vol. 11, no. 7, pp. 40:40–40:51, 2013.

[37] U. Drepper, "What every programmer should know about memory," http://people.redhat.com/drepper/cpumemory.pdf, accessed: 26/10/2017.

[38] R. Love, "Kernel Korner: CPU Affinity," *Linux J.*, vol. 2003, no. 111, pp. 8–, 2003.

[39] Linux Developers, "Migrate Pages - Move all Pages in a Process to another Set of Nodes," http://man7.org/linux/man-pages/man2/migrate_pages.2.html, accessed: 26/10/2017.

[40] J. Torrellas, H. S. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Trans. Comput.*, vol. 43, no. 6, pp. 651–663, Jun. 1994.

[41] H. Bruneel and B. G. Kim, "Scheduling Disciplines," in *Discrete-Time Models for Communication Systems Including ATM*. Boston, MA, USA: Springer US, 1993, pp. 49–74.

[42] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," in *Proc. of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, 1994, pp. 356–368.

[43] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 3rd ed. Springer Publishing Company, Incorporated, 2008.

[44] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.

[45] A. Fedorova, M. Seltzer, and M. D. Smith, "A Non-Work-Conserving Operating System Scheduler For SMT Processors," in *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, Boston, MA, USA, 6 2006.

[46] P. J. Denning, "Thrashing: Its Causes and Prevention," in *Proc. of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 915–922.

[47] S. Even, *Graph Algorithms*, 2nd ed. New York, NY, USA: Cambridge University Press, 2011.

[48] R. L. Graham, "Bounds on Multiprocessing Anomalies and Related Packing Algorithms," in *Proc. of the May 16-18, 1972, Spring Joint Computer Conference*, ser. AFIPS '72 (Spring). New York, NY, USA: ACM, 1972, pp. 205–217.

[49] ——, "Bounds on Multiprocessing Timing Anomalies," *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[50] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time, and Non-Real-Time Processes," in *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, Dec 2003, pp. 396–407.

[51] T. Lundqvist and P. Stenström, "Timing Anomalies in Dynamically Scheduled Microprocessors," in *Proc. of the 20th IEEE Real-Time Systems Symposium*, ser. RTSS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 12–.

[52] E. W. Dijkstra, "Information Streams Sharing a Finite Buffer," *Information Processing Letters*, vol. 1, no. 5, pp. 179 – 180, 1972.

[53] D. Applegate and W. Cook, "A Computational Study of the Job-Shop Scheduling Problem," *ORSA Journal on computing*, vol. 3, no. 2, pp. 149–156, 1991.

[54] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing Load Balancing and Data-Locality with Data-aware Scheduling," in *2014 IEEE International Conference on Big Data*. Washington, DC, USA: IEEE, 2014, pp. 119–128.

[55] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task Scheduling Algorithms for Heterogeneous Processors," in *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. 8th.* Washington, DC, USA: IEEE Computer Society, 1999, pp. 3–14.

[56] S. M. Imam and V. Sarkar, "Integrating Task Parallelism with Actors," *SIGPLAN Not.*, vol. 47, no. 10, pp. 753–772, Oct. 2012.

[57] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP Task Scheduling Strategies for Multicore NUMA Systems," *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 2, pp. 110–124, 2012.

[58] N. G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer*, vol. 25, no. 12, pp. 33–44, 1992.

[59] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal, "Efficient Load Balancing for Wide-area Divide-and-conquer Applications," *SIGPLAN Not.*, vol. 36, no. 7, pp. 34–43, 2001.

[60] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The Data Locality of Work Stealing," in *Proc. of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '00. New York, NY, USA: ACM, 2000, pp. 1–12.

[61] J.-N. Quintin and F. Wagner, "Hierarchical Work-stealing," in *Proc. of the 16th International Euro-Par Conference on Parallel Processing: Part I*, ser. EuroPar'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 217–229.

[62] G. Contreras and M. Martonosi, "Characterizing and Improving the Performance of Intel Threading Building Blocks," in *2008 IEEE International Symposium on Workload Characterization*, Sept 2008, pp. 57–66.

[63] E. Francesquini, A. Goldman, and J.-F. Méhaut, "Actor Scheduling for Multicore Hierarchical Memory Platforms," in *Proc. of the 12th ACM SIGPLAN Workshop on Erlang*, ser. Erlang '13. New York, NY, USA: ACM, 2013, pp. 51–62.

[64] HyperTransport Engineers, "HyperTransport Consortium," https://www.hypertransport.org/, accessed: 2017-11-13.

[65] msr-tools Developer, "msr-tools," https://01.org/msr-tools, accessed: 2017-11-13.

[66] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments," in *39th International Conference on Parallel Processing Workshops*, Sept 2010, pp. 207–216.

[67] B. Gregg, "The Flame Graph," *Commun. ACM*, vol. 59, no. 6, pp. 48–57, 2016.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, January 14, 2018 Sebastian Wölke