Work-Stealing, Locality-Aware Actor Scheduling

Saman Barghi
David R. Cheriton School of Computer Science
Univeristy of Waterloo
Waterloo, Canada
Email: sbarghi@uwaterloo.ca

Martin Karsten
David R. Cheriton School of Computer Science
Univeristy of Waterloo
Waterloo, Canada
Email: mkarsten@uwaterloo.ca

***Abstract*—The actor programming model is gaining popularity due to the prevalence of multi-core systems along with the rising need for highly scalable and distributed applications. Frameworks such as Akka, Orleans, Pony, and C++ Actor Framework (CAF) have been developed to address these application requirements. Each framework provides a runtime system to schedule and run millions of actors, potentially on multi-socket platforms with non-uniform memory access (NUMA). However, the literature provides only limited research that studies or improves the performance of actor-based applications on NUMA systems. This paper studies the performance penalties that are imposed on actors running on a NUMA system and characterizes applications based on the actor type, behavior, and communication pattern. This information is used to identify workloads that can benefit from improved locality on a NUMA system. In addition, two locality- and NUMA-aware work-stealing schedulers are proposed and their their respective execution overhead in CAF is studied on both AMD and Intel machines. The performance of the proposed work-stealing schedulers is evaluated against the default scheduler in CAF.**

***Keywords*-Actors, Scheduling, NUMA, Locality**

I. Introduction

Modern computers utilize multi-core processors to increase performance, because the breakdown of Dennard scaling [1] makes substantial increase of clock frequencies unlikely and because the depth and complexity of instruction pipelines have also reached a breaking point. Other hardware scaling limitations have led to the emergence of non-uniform memory access (NUMA) multi-chip platforms [2] as a trade-off between low-latency and symmetric memory access. Multi-core and multi-chip computing platforms provide a shared memory interface across a non-uniform memory hierarchy by way of hardware-based cache coherence [3].

The actor model of computing [4], [5], [6] is a model for writing concurrent applications for parallel and distributed systems. The actor model provides a high-level abstraction of concurrent tasks where information is exchanged by message passing, in contrast to task parallelism where tasks communicate by sharing memory. A fundamental building block of any software framework implementing the actor model is the proper distribution and scheduling of actors on multiple underlying processors (often represented by kernel/system threads) and the efficient use of system resources to achieve the best possible runtime performance. The C++ Actor Framework (CAF) [7] provides a runtime for the actor model and has a low memory footprint and improved CPU utilization to support a broad range of applications.

This paper studies the challenges for a popular type of actor scheduler in CAF, *work-stealing*, on NUMA platforms. Actor models have specific characteristics that need to be taken into account when designing scheduling policies. The contributions are: a) the structured presentation of these characteristics, b) an improved hierarchical scheduling policy for actor runtime systems, c) the experimental evaluation of the new scheduling proposals and study their performance in comparison to a randomized work-stealing scheduler.

The rest of the paper is organized as follows. The next section provides background information related to scheduling, the actor programming model, and the C++ Actor Framework (CAF). Section 3 describes existing research work related to the problem studied in this paper, while Section 4 presents a discussion of workload and application characteristics, followed by the actual scheduling proposal. An experimental evaluation is provided in Section 5 and the paper is concluded with brief remarks in Section 6.

II. Background

A. The Actor Programming Model

In the actor model the term *actor* describe autonomous objects that communicate asynchronously through messages. Each actor has a unique address that is used to send messages to that actor, and each actor has a mailbox that is used to queue received messages before processing. Actors do not share state and only communicate by sending messages to each other. Sending a message is a nonblocking operation and an actor processes each message in a single atomic step. Actors may perform three types of action as a result of receiving a message: *(1) send messages to other actors, (2) create new actors, (3) update their local state* [6], [8]. Actors can change their behavior as a result of updating their local state. In principle, message processing in a system of actors is non-deterministic, because reliable, in-order delivery of messages is not guaranteed. This non-deterministic nature of actors makes it hard to predict their behavior based on static compile-time analysis or dynamic analysis at runtime.

Despite their differences [9], all actor systems provide a runtime that multiplexes actors onto multiple system threads to take advantage of multi-core hardware and provide concurrency and parallelism. For example, Erlang [10] uses a virtual machine along with a work-stealing scheduler to distribute actors and to balance the workload. Akka [11] provides various policies to map actors to threads and by default uses a fork/join policy. Pony [12] provides only a work-stealing scheduler, while CAF provides both work-sharing and work-stealing policies. The type of the actor runtime can influence how tasks should be scheduled in regard to locality. For example, in a managed language such as Erlang actors have their own private heap and stack whereas in CAF actor objects and variables are directly allocated from the default global heap.

Scheduling is affected by implementation details of the actor framework and the type of workload. Most importantly, depending on the type of an actor and how memory is allocated and accessed by that actor, scheduling might or might not benefit from locality improvements related to CPU caches or NUMA. Therefore, it is important to identify all factors at play when it comes to locality-aware scheduling. These factors must be studied both individually and in combination to determine scenarios that benefit from locality-aware scheduling in a message-passing software framework.

B. Actor Model vs. Task Parallelism

Actor-based systems can benefit from work-stealing due to the dynamic nature of tasks and their asynchronous execution, which is similar to task parallelism. However, most variations of task parallelism, e.g., dataflow programming, are used to deconstruct strict computational problems to exploit hardware parallelism. As such, interaction patterns between tasks are usually deterministic, because dependencies betweens tasks are known at runtime. Tasks are primarily concerned with the availability of input data and therefore do not have any need to track their state. In contrast, the actor model provides nondeterministic, asynchronous, message-passing concurrency. Computation in the actor model cannot be considered as a directed acyclic graph (DAG) [7] (e.g., Cilk assume DAG computation through fork/join [13]) and actors usually have to maintain state. Due to these differences, the internals of an actor runtime, such as scheduling, differ from runtime systems aimed at task parallelism.

Most importantly, applications written using the actor model, such as a chat server, are sensitive to latency and fairness for individual tasks. Therefore, a scheduler designed for an actor system must be both efficient and fair, otherwise applications show a long tail in their latency distribution. On the contrary, for task parallelism, as long as the the entire problem space is explored in an acceptable time, fairness and latency of individual tasks does not matter.

Furthermore, in task parallelism many lightweight tasks are created that run from start to end without yielding execution. In contrast, actors in an actor system can wait for events and other messages, or cooperatively yield execution to guarantee fairness. Hence, the execution pattern of actors is different from tasks in a task-parallel workload. This affects both scheduling objectives and locality.

Finally, since actors fully encapsulate their internal state and only communicate through passing messages that are placed into the respective mailboxes of other actors, the resulting memory access pattern is not necessarily the same as the access pattern seen in task-parallel frameworks. For instance, in OpenStream [14] each consumer task has multiple data-buffers for each producer task. OpenMP 4.0 [15] allows task dependencies through shared memory, however this is based on a deterministic sporadic DAG model which only allows dependencies to be defined among sibling tasks.

Therefore, although locality-aware scheduling is a well-studied topic for task parallelism, due to those differences, it cannot automatically be assumed that findings for task parallelism directly translate into similar findings for actor models - or vice versa.

C. Work-Stealing Scheduling

Multiprocessor scheduling is a well-known NP-hard problem. In practice, runtime schedulers apply broad strategies to satisfy objectives, such as resource utilization, load balancing, and fairness. *Work-stealing* has emerged as a popular strategy for task placement and also load balancing [16]. Work-stealing primarily addresses resource utilization by stipulating that a processor without work "steals" tasks from another processor.

Work-stealing has been investigated for general multithreaded programs with arbitrary dependencies [17], generalizing from its classical formulation limited to the fork-join pattern. The main overhead of work-stealing occurs during the stealing phase when an idle processor polls other deques to find work, which might cause interprocessor communication and lock contention that negatively impact performance. The particulars of victim selection vary among work-stealing schedulers. In Randomized Work-Stealing (RWS), when a worker runs out of work it chooses the victims randomly. The first randomized work-stealing algorithm for fully-strict computing is given in [17]. The algorithm has an expected execution time of $T_1/P + O(T_\infty)$ on P processors, and also has much lower communication cost than work-sharing.

For message-driven applications, such as those built with actor-based programming, these bounds can only be regarded as an approximation. The reason is that deep recursion does not occur in event-based actor systems, since computation is driven by asynchronous message passing and cannot be considered as a DAG.

It has been shown that work-stealing fundamentally is efficient for message-driven applications [18]. However, random victim selection is not scalable [19], because it does not take into account locality, architectural diversity, and the memory hierarchy [18], [20], [21]. In addition, RWS does not consider data distribution and the cost of inter-node task migration on NUMA platforms [21], [22], [23].

D. C++ Actor Framework

The C++ Actor Framework (CAF) [7] provides a runtime that multiplexes N actors to M threads on the local system. The number of threads (M) is configurable and by default is equal to the number of cores available on the system, while the number of actors (N) changes during runtime. Actors in CAF transition between four states: *ready*, *waiting*, *running*, and *done*. An actor changes its state from *waiting* to *ready* in reaction to a message being placed in its mailbox. Actors in CAF are modeled as lightweight state machines that are implemented in user space and cannot be preempted.

CAF's work-stealing scheduler uses a double-ended task queue per worker thread. Worker threads treat this deque as LIFO and other threads treat the queue as FIFO. New tasks that are created by an actor are pushed to the head of the local queue of the worker thread where the actor is running. Tasks that are created by spawning actors outside of other actors, e.g., from the `main()` function, are placed into the task queues in a round robin manner for load balancing.

Actors create new tasks by either spawning a new actor or sending a message to an existing actor with an empty mailbox. If the receiver actor's mailbox is not empty, sending a message to its mailbox does not result in creation of a new task since a task already processes the existing messages.

The RWS scheduler in the CAF runtime uses a uniform random number generator to randomly select victims when a worker thread runs out of work. Although CAF provides a support layer for seamless heterogeneous hardware to bridge architectural design gaps between platforms, it does not yet provide a locality-aware work-stealing scheduler.

E. NUMA Effects

Contemporary large-scale shared-memory computer systems are built using non-uniform memory access (NUMA) hardware where each processor chip has direct access to a part of the overall system memory, while access to the remaining remote memory (which is local to other processor chips) is routed through an interconnect and thus slower. NUMA provides the means to reach a higher core count than single-chip systems, albeit at the (non-uniform) cost of occasionally accessing remote memory. Clearly, with remote memory, the efficacy of the CPU caching hierarchy becomes even more important. However, CPU caches are shared between cores with the particular nature of sharing depending on the particulars of the hardware architecture. Therefore, new scheduling algorithms are proposed that are aware of the memory hierarchy and shared resources to exploit cache and memory locality and minimize overhead [16], [20], [21], [22], [23], [24], [25].

Accordingly, efficient scheduling of actors on NUMA machines requires careful analysis of applications built using actor programming. Application analysis must be combined with a proper understanding of the underlying memory hierarchy to limit the communication overhead, scheduling costs, and achieve the best possible runtime performance.

III. Related Work

There has been very limited research addressing locality-aware or specifically NUMA-aware scheduling for actor runtime systems. Francesquini et al. [22] provide a NUMA-aware runtime environment based on the Erlang virtual machine. They identify *actor lifespan* and *communication cost* as information that the actor runtime can use to improve performance. Actors with a longer lifespan that create and communicate with many short-lived actors are called *hub* actors. The proposed runtime system lowers the communication cost among actors and their hub actor by placing the short-lived actors on the same NUMA node as the *hub* actor, called *home node*. When a worker thread runs out of work, it first tries to steal from workers on the same NUMA node. If unsuccessful, the runtime system tries to migrate previously migrated actors back to that home node. The private heap of each actor is allocated on the home node, so executing on the home node improves locality. As a last resort the runtime steals actors from other NUMA nodes and moves them to the worker's NUMA node.

Although the evaluation results look promising, the caveat, as stated by the authors, is in assuming that hub actors are responsible for the creation of the majority of actors. This is a strong assumption only applies to some applications. Also, when multiple hub actors are responsible for creating actors, the communication pattern among none-hub actors can still be complicated. Another assumption is that all NUMA nodes have the same distance from each other, and the scheduler does not take the CPU cache hierarchy into account. The approach presented takes advantage of knowledge that is available within the Erlang virtual machine, but not necessarily available in an unmanaged language runtime, such as CAF.

In contrast, the work presented here is not based on any assumptions about the communication pattern among actors or information available through a virtual machine runtime. Instead, it is solely focused on improving performance by improving the scheduler. Also, the full extent and variability of the memory hierarchy is taken into account.

A simple *affinity*-type modification to task scheduling is reported in [26]. Tasks in this system are blocking on a channel waiting for messages to arrive, and thus show similar behavior to actors waiting on empty mailboxes. In contrast to basic task scheduling, an existing task that is unblocked is never placed on the local queue. Instead, it is always placed at the end of the queue of the worker thread previously executing the task. A task is only migrated to another worker thread by stealing. This modification leads to significant performance improvements for certain workloads and thus contradicts assumptions about treating the queues in LIFO manner. However, the system has a well-defined communication pattern and long task lifespans, in contrast to an actor system that is non-deterministic with a mixture of short- and long-lived actors. We have implemented both a LIFO and an affinity policy using our hierarchical scheduler and present results in Section V.

Various locality-aware work-stealing schedulers have been proposed for other parallel programming models and shown to improve performance. Suksompong et al. [25] investigate localized work-stealing and provide running time bounds when workers try to steal their own work back. Acar et al. [27] study the data locality of work-stealing scheduling on shared-memory machines and provide lower and upper bounds for the number of cache misses, and also provide a locality-guided work-stealing scheduling scheme. Chen et al. [28] present a cache-aware two-tier scheduler that uses an automatic partitioning method to divide an execution DAG into the inter-chip tier and the intra-chip tier.

Olivier et al. [29] provide a hierarchical scheduling strategy where threads on the same chip share a FIFO task queue. In this proposal, load balancing is performed by work sharing within a chip, while work-stealing only happens between chips. In follow-up work, the proposal is improved by using a shared LIFO queue to exploit cache locality between sibling tasks as well as between a parent and newly created task [23]. Moreover, the work-stealing strategy is changed, so that only a single thread can steal work on behalf of other threads on the same chip to limit the number of costly remote steals. Pilla et al. [30] propose a hierarchical load balancing approach to improve the performance of applications on parallel multi-core systems and show that Charm++ can benefit from such a NUMA-aware load balancing strategy.

Min et al. [21] propose a hierarchical work-stealing scheduler that uses the *Hierarchical Victim Selection (HVS)* policy to determine from which thread a thief steals work, and the *Hierarchical Chunk Selection (HCS)* policy that determines how much work a thief steals from the victim. The HVS policy relies on the scheduler having information about the memory hierarchy: cache, socket, and node (this work also considers many-core clusters). Threads first try to steal from the nearest neighbors and only upon failure move up the locality hierarchy. The number of times that each thread tries to steal from different levels of the hierarchy is configurable. The victim selection strategy presented here in Section IV-B is similar to HVS, but takes NUMA distances into account.

Drebes et al. [31], [32] combine topology-aware work-stealing with work pushing and dependence-aware memory allocation to improve NUMA locality and performance for data-flow task parallelism. Work pushing transfers a task to a worker whose node contains the task's input data according to some dependence heuristics. Each worker has a Multiple-Producer-Single-Consumer (MPSC) FIFO queue in addition to a work-stealing dequeue. The MPSC queue is only processed when the deque is empty. However, this approach is not applicable to latency-sensitive actor application for two reasons: first, the actor model is nondeterministic and data dependence difficult to infer at runtime. Second, adding a lower-priority MPSC queue adds complexity and can cause some actors to be inactive for a long time, which violates fairness and thus causes long tail latencies for the application. Moreover, the proposed deferred memory allocation relies on knowing the task dependencies in advance, which is not possible with the actor model. Therefore, this optimization cannot be applied to the actor model. The topology-aware work-stealing introduced by this work is similar to ours, but it is evaluated in combination with deferred memory allocation and work-pushing. Thus, it is not possible to discern the isolated contribution of topology-aware work-stealing.

Majo et al. [2] specify that optimizing for data locality can counteract the benefits of cache contention avoidance and vice versa. In Section V we present results that demonstrate this effect for actor workloads where aggressive optimization for locality increases the last-level cache contention.

IV. Locality-Aware Scheduler for Actors

This section first discusses the key characteristics of an actor-based application and workload that need to be considered when designing a locality-aware, work-stealing scheduler. These characteristics also provide valuable hints for designing evaluation benchmarks. Based on these findings, a novel hierarchical, locality-aware work-stealing scheduler is presented.

A. Characteristics of Actor Applications

Key operations can be slowed down when an actor migrates to another core on a NUMA system, depending on the NUMA distance. This performance degradation can come from messages that arrive from another core, or from accessing the actor's state that is allocated on different NUMA node. Depending on the type of actor and the communication pattern, the amount of degradation differs. Therefore, improving locality does not benefit all workloads. We identify the following factors in applications and workloads for actor-based programming that can affect the performance of a work-stealing scheduler on a hierarchical NUMA machine:

1) Memory allocated for actor and access pattern: Actors sometimes only perform computations on data passed to them through messages. For simplicity, we denote actors that only depend on message data as *stateless* actors, and actors that do manage local state as *stateful* actors.

Stateful actors allocate local memory upon creation and access it or perform other computations depending on the type of a message and their state when they receive a message. A stateful actor with sizable state and intensive memory access to that state is better executed closer to the NUMA node where it was created. Also, for better cache locality, especially if the actor receives messages from its spawner frequently, it is better to keep such an actor on the same core, or a core that shares a higher-level CPU cache with the spawner core. The reason is that those messages are hot in the cache of the spawner actor.

On the other hand, stateless actors do not allocate any local memory and can be spawned multiple times for better scalability. For such actors, the required memory to process messages is allocated when they start processing a message and deallocated when the processing is done. Therefore, the only substantial memory that is accessed is the one allocated for the received message by the sender of that message. Such actors are better executed closer to the message sender.

2) Message size and access pattern: The size of messages has a direct impact on the performance and locality of actors on NUMA machines. Messages are allocated on the NUMA node of the sender, but accessed on the core that is executing the receiver actor. If the size of messages is typically larger than the size of the local state of an actor, and the receiving actor accesses the message intensively, actors are better to be activated on the same node as the sender of the message.

3) Communication pattern: Since the actor model is non-deterministic, it is difficult to generally analyze the communication pattern between actors. Two actors that are sending messages to each other can go through different states and thus have various memory access patterns. In addition, the type and size of each message can vary depending on the state of the actor. We do not make any assumptions about the communication pattern of actors, unlike others [22].

Aside from illustrating the trade-offs involved in actor scheduling, these observations are also useful to determine which benchmarks realistically demonstrate the benefits of locality-aware work stealing schedulers and which represent worst-case tests.

B. Locality-Aware Scheduler (LAS)

Our locality-aware scheduler consists of three stages: memory hierarchy detection, work placement, and work stealing. When an application starts running, the scheduler determines the memory hierarchy of the machine. Also, a new actor is is placed on the local or a remote NUMA node depending on the type of the actor. Finally, when a worker thread runs out of work it uses a locality-aware policy to steal work from another work thread.

1) Memory Hierarchy Detection: Our work-stealing algorithm needs to be aware of the memory hierarchy of the underlying system. In addition to the cache and NUMA hierarchy, differing distances between NUMA nodes are an important factor in deciding where to steal tasks from. Access latencies can vary significantly based on the topological distance between access node and storage node.

The scheduler builds a representation of the locality hierarchy using the *hwloc* library [33] that uses hardware information to determine the memory hierarchy, which the scheduler represents as a tree. The root is a place-holder representing the whole system, while intermediate levels represent NUMA nodes and groups, taking into account NUMA distances. Subsequent nodes represent shared caches and the leaves represent the cores on the system. This approach is independent from any particular hardware architecture.

2) Actor Placement: For fully-strict computations, data dependencies of a task only go to its parent. Thus the natural placement for new tasks the core of the parent task. However, actors can communicate arbitrarily and thus, local placement of newly created actors does not guarantee the best performance. For example, actors receiving remote messages pollute the CPU cache for actors that execute later and process messages from their parents. Also, as stated earlier, depending on the size of the message in comparison to state variables, placing the actor in the sender's NUMA node can help or hurt performance. Determining the best strategy at runtime can add significant overhead, so there is no apparent optimal approach.

The exception are *hub actors* [22], i.e., long-living actors that spawn many children and communicate with them frequently. Such actors place high demand on the memory allocator and can interfere with each other if placed on the same NUMA node. Furthermore, if a locality-aware affinity policy tries to keep actors on their home node, placing multiple hub actors on the same NUMA node further increases contention over shared resources and thus reduces the performance. Hence, our scheduler uses the same algorithm for initial placement of *hub actors* [22] to spread them across different NUMA nodes. The programmer needs to annotate hub actors. The system then tags corresponding structures at compile time and the runtime scheduler uses this information to place such actors far from each other.

3) Locality-aware Work-stealing: A locality-aware victim selection policy attempts to keep tasks closer to the core that created them or was running them previously to take advantage of better cache locality. Depending on the hardware architecture, cores might share higher-level CPU cache. Therefore, in our scheduler the thief worker thread first steals from worker threads executing on nearby cores in the same NUMA node with shared caches to improve locality. If there is no work available in the local NUMA node, the hierarchical victim selection policy tries to steal jobs from worker threads of other NUMA nodes with increasing NUMA distance. The goal of NUMA-aware work stealing is to avoid migrating actors between NUMA nodes to the extent possible, and thus to remove the need for remote memory accesses.

Limiting the worker threads to initially choose their victims within their own NUMA node can lead to more frequent contention over deques on the local NUMA node in comparison to using the random victim selection strategy. For example, if a single queue still has work, while all other worker threads run out of work, is the worst-case scenario for a work-stealing scheduler. However, this case appears frequently in actor applications, where a hub actor creates multiple actors and other worker threads steal from the local deque of the thread that runs the hub actor. Our investigation shows that when stealing fine-grained tasks with workloads that are 20μs or shorter, the performance penalty ratio increases exponentially as the number of thief threads increase. For more coarse-grained tasks, the performance penalty is not significant, since the probability of contention decreases.

To alleviate this problem, our scheduler keeps track of the number of threads per NUMA node that are polling the local node. This number is used along with the approximate size of the deques in the node to reduce the number of threads that are simultaneously polling a deque (Algorithm 1). If there is only a single non-empty deque and more than half of threads under that node are polling that deque, the thief thread backs off and tries again later.

In addition, polling the queue of many other worker threads with empty queues can result in wasting CPU cycles when the number of potential victims is limited. In CAF, a worker thread constantly polls its own deque and after a certain number of attempts, polls a victim deque. To avoid wasting CPU cycles, we have modified the deque and added an approximate size of the deque using a counter. A thief uses this approximate size when it attempts to steal from other workers executing on the same NUMA node. If there are non-empty queues, it chooses one randomly, otherwise if all the queues are empty, the thief immediately moves up to the next higher level (Algorithm 1). This approach removes the overhead of polling empty queues on the local NUMA node and thus decreases the number of wasted CPU cycles. Since there is a fixed number of cores on a NUMA node, scanning their queue sizes adds little overhead that remains constant even when the application scales.

When a worker runs out of work, it becomes a thief and uses the memory hierarchy tree provided by the scheduler to perform hierarchical victim selection as described in Algorithm 1. The updated vertex v is passed to the function each time to complete the tree traversal. An empty result or a victim with an empty deque means that the thief has to try again.

We have created two variants of LAS that differ in their placement strategy. When an existing actor is unblocked by a message, the *local* variant (LAS/L) places the actor on the local deque, while the *affinity* variant (LAS/A) places the actor at the end of the deque of the worker thread previously executing it. In both cases, newly created actors are pushed to the head of the local deque. LAS/L is similar to typical work-stealing placement where all activated and newly created tasks are pushed to the head of the local deque. LAS/A improves actor-to-thread (and thus to-core) affinity,

Algorithm 1 Hierarchical Victim Selection

```
1:  T: Memory hierarchy tree
2:  C: Set of cores under v
3:  p: Number of threads polling under local NUMA node
4:  r: Number of steal attempts for v
5:  procedure ChooseVictim(v)
6:      if $r = Size(C)$ and $v \neq \text{root}(T)$ then
7:          $v \leftarrow \text{parent}(v)$
8:      if $v$ is in local NUMA node then
9:          $S = \{s \mid \text{all non-empty local deques}\}$
10:         if $S = \varnothing$ then
11:             $v \leftarrow \text{parent}(v)$
12:         else if $\text{size}(S) = 1$ and $p > \frac{size(C)}{2}$ then
13:             return $\varnothing$
14:         else return random from $S$
15:     return random from $C$
```

because actors are moved only by stealing. However, it adds overhead to saturated workers and increases contention when placing actors on remote deques, which is further discussed in Section V-C.

V. Experiments and Evaluation

Experiments are conducted on an Intel and an AMD machine that have different NUMA topologies and memory hierarchies. The Intel machine is a Xeon with 4 sockets, 4 NUMA nodes, and 32 cores. Each NUMA node has 64 GB of memory for a total of 256 GB. Each socket has 8 cores, running at 2.3 GHz, that share 16 MB L3 Cache, and each core has private L1 and L2 caches. Each NUMA node is only directly connected to two other NUMA nodes. Hpyer-threading is disabled. The AMD machine is an Opteron with 4 sockets, 8 NUMA nodes, and 64 cores. Each NUMA node contains 64 GB of memory for total of 512 GB. Each socket has 8 cores running at 2.5 GHz. Each core has a private L1 data cache, and shares the L1 instruction cache and an L2 cache (2 MB) with a neighbour core. All cores in the same socket share one L3 cache (6 MB). The experiments are performed with CAF version 0.12.2, compiled with GCC 5.4.0, on Ubuntu Linux 16.04 with kernel 4.4.0.

The experiments compare CAF's default Randomized Work-Stealing (RWS) scheduler with LAS/L and LAS/A. We first evaluate the performance using benchmarks to study the effect of scheduling policy on different communication patterns and message sizes. Next, we use a simple chat server to observe the efficiency of schedulers for an application that has a large number of actors with non-trivial communication patterns, different behaviors, and various message sizes.

A. Benchmarks

The first set of experiments attempts to isolate the effects of each scheduling policy for different actor communication patterns. A subset of benchmarks from the BenchErl [34] and Savina [35] benchmark suites is chosen that represents typical communication patterns used in actor-based applications. Some of these benchmarks are adopted from task-parallelism benchmarks, but modified to fit the actor model.

- *Big (BIG):* In a many-to-many message passing scenario many actors are spawned and each one sends a ping message to all other actors. An actor responds with a pong message to any ping message it receives.
- *Bang (BANG):* In a many-to-one scenario, multiple senders flood the one receiver with messages. Senders send messages in a loop without waiting for any response.
- *Logistic Map Series (LOGM):* A synchronous request-response benchmark pairs control actors with compute actors to calculate logistic map polynomials through a sequence of requests and responses between each pair.
- *All-Pairs Shortest Path (APSP):* This benchmark is a weighted graph exploration application that uses the Floyd-Warshall algorithm to compute the shortest path among all pairs of nodes. The weight matrix is divided into blocks. Each actor performs calculations on a particular block and communicates with the actors holding adjacent blocks.
- *Concurrent Dictionary (CDICT):* This benchmark maintains a key-value store by spawning a dictionary actor with a constant-time data structure (hash table). It also spawns multiple sender actors that send write and read requests to the dictionary actor. Each request is served with a constant-time operation on the hash table.
- *Concurrent Sorted Linked-List (CSLL):* This benchmark is similar to CDICT but the data-structure has linear access time (linked list). The time to serve each request depends the type of the operation and the location of the requested item. Also, actors can inquire about the size of the list, which requires iterating through all items.
- *NQueens first N Solutions (NQN)*: A divide-and-conquer style algorithm searches a solution for the problem: "How can N queens be placed on an $N \times N$ chessboard, so that no pair attacks each other?"
- *Trapezoid approximation (TRAPR)*: This benchmark consists of a master actor that partitions an integral and assigns each part to a worker. After receiving all responses they are added up to approximate the total integral. The message size and computation time is the same for all workers.
- *Publish-Subscribe (PUBSUB)*: Publish/subscribe is an important communication pattern in actor programs that is used extensively in many applications, such as chat servers and message brokers. This benchmark is implemented using CAF's group communication feature and measures the end-to-end latency of individual messages. It represents a one-to-many communication pattern where a publisher actor sends messages to multiple subscribers. Actors can subscribe to more than one publisher.

B. *Experiments*

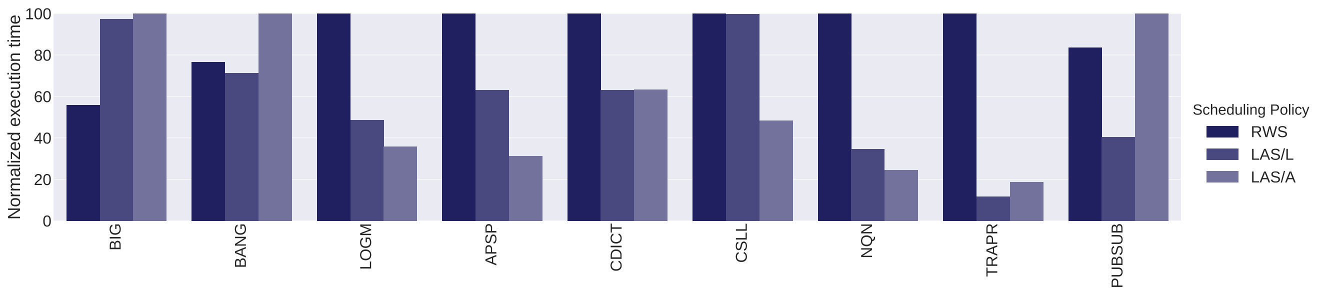
The benchmark results are shown in Figure 1. The execution time is the average of 10 runs and normalized to the slowest scheduler. All experiments are configured to keep all cores busy most of the time, i.e., the system operates at peak load.

The RWS scheduler performs relatively better for the BIG benchmark and it outperforms both LAS/L and LAS/A. This benchmark represents a symmetric many-to-many communication pattern where all actors are sending messages to each other. This workload benefits from a symmetric distribution of work. Other experiments (not shown here) show that using the NUMA *interleave* memory allocation policy improves the performance further. For this particular workload, improving locality does not translate to improving the performance.

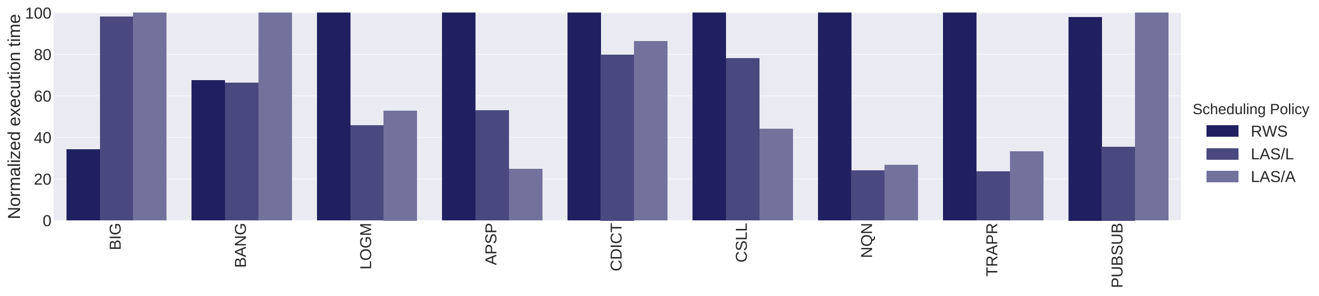
The BANG benchmark represents workloads using many-to-one communication. Messages have very small sizes and no computation is performed. Since the receiver's mailbox is the bottleneck, improving locality does not significantly affect the overall performance. LAS/L only improves the performance slightly by allocating more messages on the local node.

LOGM and APSP both create multiple actors during startup and each actor frequently communicates with a limited number of other actors. In addition, computation depend on an actor's local state and message content. For both workloads, LAS/L and LAS/A outperform RWS by a great margin. In such workloads, each actor can only be activated by one of the actors it communicates with. If one of the communicating actors is stolen and executes on another core, in RWS and LAS/L it causes the other actors to follow and execute on the new core upon activation. Since all actors maintain local state that is allocated upon creation of the actor, all actors that are part of the communication group experience longer memory access times if one of them migrates to another NUMA node. Keeping actors on the same NUMA node and closer to the core they were running before can prevent this. LAS/A improves performance further by preventing other actors to migrate along with the stolen actor. Even though actors are occasionally moved to another NUMA node, the rest of the group stays on their own NUMA node. Thus, LAS/A performs better than LAS/L by preventing group migration of actors. For LOGM, since each pair of actors is isolated from other actors, stealing one actor translates to moving one other actor along with it. However, in APSP each actor communicates with multiple actors, which means stealing one actor can cause a chain reaction and several actors that do not directly communicate with the stolen actor might also migrate. LAS/A therefore has a stronger effect on APSP than LOGM.

CDICT and CSLL represent workloads where a central actor spawns multiple worker actors and communicates with



(a)



(b)

Figure 1. Results of running the benchmarks with various schedulers on (a) AMD Opteron and (b) Intel Xeon. The results are normalized to slowest scheduler (lower is better).

them frequently. The central actor is responsible for managing a data structure and receives read and write requests from the worker actors. CDICT benefits from improved locality provided by both locality-aware schedulers. Since the majority of operations are allocating and accessing messages between the central actor and the worker actors, placing worker actors closer to the central actor leads to improved performance due to faster memory accesses. In CDICT all requests are served from a hash table in roughly constant time. In such a setting, LAS/A can cause an imbalance in service times, because some actors are being placed on cores with higher memory access times. Since the service time for each request is fairly small, this additional overhead can slightly slow down the application.

However, in CSLL LAS/A outperforms LAS/L and RWS. First, the overhead that LAS/A imposes on the central actor becomes negligible in comparison with the linear lookup time into the linked list. Because of the resulting increased service times, most actors ultimately become inactive, waiting for a response from the central actor. The corresponding worker threads end up being idle and seeking work. With LAS/A, the response unblocks a worker actor on its previous worker thread, so that execution can continue right away. However, LAS/L unblocks worker actors on the same worker thread as the central actor. This introduces additional latency until the worker actor executed or alternatively, until it is stolen by an idle worker thread.

NQN is a divide-and-conquer algorithm where a master actor is responsible for dividing the work among fixed number of worker actors. Each worker actor performs a recursive operation on the task assigned to it and further divides the task to smaller subtasks. But instead of spawning new actors, it reports back to the master actor that assigns the new tasks to the worker actors in a round-robin fashion. Therefore, all worker actors are constantly producing and consuming messages. The computation performed for each message depends on the content of the message and all items in each message are accessed during computation. Improving locality and placing worker actors closer to each other and to the master actor has a significant impact on performance. LAS/L and LAS/A perform 5 times faster than RWS in this case.

In TRAPAR worker actors receive a message from a master actor, perform some calculations, send back a message, and exit. LAS/L and LAS/A improve the performance up to 10 times for this benchmark. Since all actors are created on the local deque of the master actor, and the tasks are very fine-grained, locality-aware scheduling increases the chance of local cores to steal and run these tasks closer to the master actor. Since all communications are with the master actor, the performance is improved significantly.

The PUBSUB benchmark shows significant end-to-end message latency improvement when LAS/L policy is used in comparison with RWS. LAS/L keeps the subscribers closer to the publisher that sends them a message and improves the locality. However, LAS/A shows worse performance than

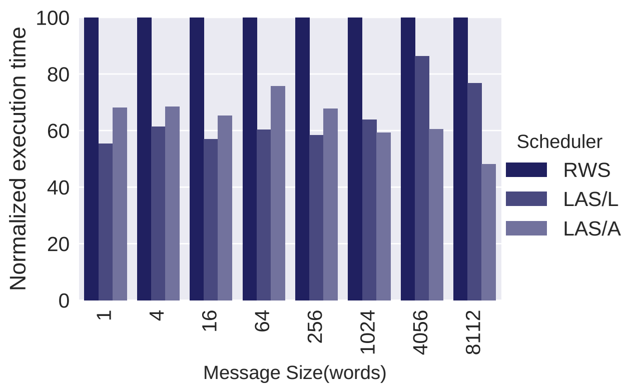


Figure 2. Results for varying the size of the *value* in CDICT benchmark. The execution times are normalized to the slowest scheduler (lower is better)

the other two policies. Profiling the code reveals that worker threads are stalled by lock contention most of the time. The reason is that with LAS/A, worker threads place the newly created tasks on the deque of other cores rather than the local core. Since publishers are constantly unblocking actors on other cores, this leads to higher contention when there are large number of publishers and subscribers.

There are minor differences between the results from the AMD machine and the Intel machine. These differences come from the differences in the NUMA setup of each machine, explained at the beginning of this section. The probability that RWS moves tasks to a NUMA node with higher access times is higher for the AMD machine. Thus, locality-aware schedulers are slightly more effective on the AMD machine.

In general, the results indicate that workloads with many-to-many communication patterns (BIG) do not benefit from locality-aware schedulers. On the other hand, workloads where actors are communicating with a small cluster of other actors (LOGM and APSP), actors communicate with a central actor and access message contents (CDICT and CSLL), or actors communicate with a central actor one or multiple times and perform computations that depend on the content of the message (NQN and TRAPR), benefit from locality-aware schedulers. Moreover, in most cases where locality improves performance, LAS/A performs similar or better in comparison with LAS/L. However, in one case (PUBSUB) LAS/A causes high contention and a performance decrease.

We have performed another experiment to study the effect of message size on the performance of locality-aware schedulers. The CDICT benchmark is modified to make the *value* size configurable for each key-value pair. This affects the size of messages and the size of memory operations performed by the central actor. Worker actors submit write requests 20% of the time. Figure 2 shows the results for this experiment executed on the AMD machine. Experiments on the Intel machine show similar results, but are not shown here due to limited space.

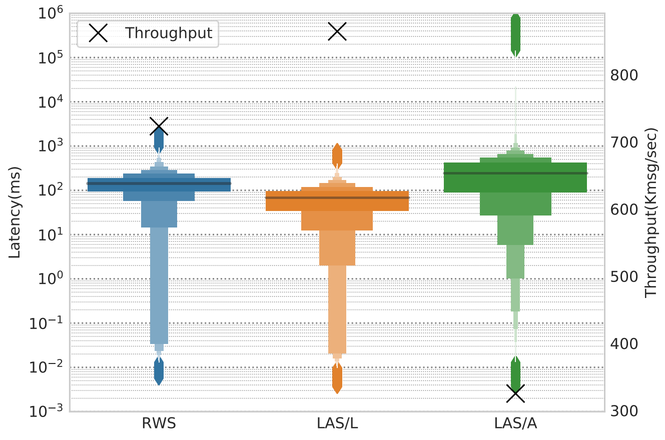
The results show that the LAS/L scheduler outperforms both RWS and LAS/A for value sizes smaller than 256 words. LAS/L improves locality and since most messages and objects fit into lower level caches (L1 and L2), improved locality further improves the performance. RWS distributes tasks among NUMA nodes and therefore imposes higher memory access times. LAS/A also adds additional overhead, because it causes the dictionary actor to unblock some actors on remote NUMA nodes. However, as the value size gets larger, messages and objects do not fit into lower level caches. Since LAS/L keeps most actors on the same NUMA node as the dictionary actor, this creates contention in the L3 cache, which slows down the dictionary actor. LAS/A, on the other hand, distributes actors to other NUMA nodes as well, which avoids the contention in L3, such that remote access overhead is compensated by lower contention. RWS also avoids the contention problem and therefore the difference between LAS/L and RWS decreases. In fact, when increasing the percentage of write requests, we have observed that RWS can even outperform LAS/L (not shown due to limited space).

C. Chat Server

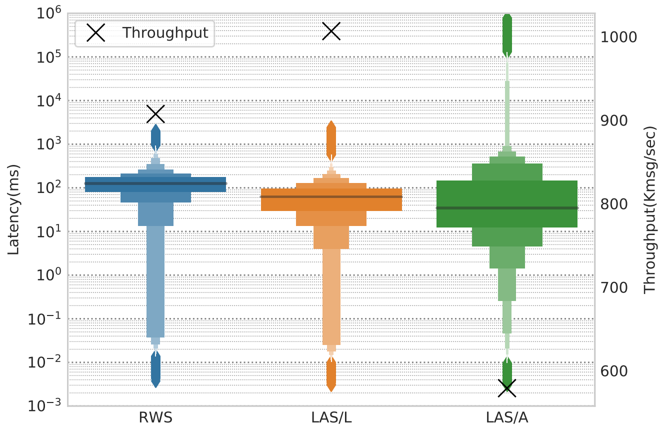
To evaluate both variants of LAS using a more realistic scenario, we have implemented a chat server similar to [36] that supports one-to-one and group chats. Each user (session) is represented by an actor that holds the state for the session in the server application. Chat groups are created using the publish/subscribe based group communication in CAF. To simplify the implementation the server does not include network operations and the workload is generated and consumed in the same process. However, the chat server implements pre- and post-processing operations that would normally be carried out in the context of communication with remote clients, such as encryption.

Each user has a friend list, group list, and blocked list, which represent the corresponding lists of users respectively. Information about each session and a log of messages is stored in an in-memory key-value storage controlled by a database actor. In addition, each session actor stores its information in a local cache controlled by a local cache actor. When a session actor receives a message, it first decrypts the message, uses the receiver user ID to find the reference to the receiving actor, and forwards the message to that actor. The message is also logged in both the local cache and the central storage. When the receiver actor receives the message, it first checks whether the sender is in its blocked list. If not, it encrypts the message as if it was sent out to a remote client. If a message is sent to a group, an actor representing the group forwards the message to all subscribers, which creates a one-to-many communication pattern.

The chat server is configured to run with 1 million actors and 10000 groups. Each user has random number of



(a)



(b)

Figure 3. Throughput and distribution of end-to-end message latency using different scheduling policies on (a) AMD Opteron and (b) Intel Xeon. Latencies are quantified on the left Y-axis that a logarithmic scale (lower is better). Throughput is quantified on the 2nd Y-axis (Higher is better).

friends (max. 100), subscribes to a random number of groups (max. 10), and blocks random number of users (max. 5). Random numbers uniformly distributed and all session and group actors are spawned before the experiment starts. To simulate receiving messages from users, every second each session actor uses a timer to send a message with a random length up to 1024 bytes to a randomly chosen user or group from its friend or group list. 5% of the messages are sent to groups and the rest are direct messages. The experiment drives each system to peak load. We measure the overall throughput, along with the end-to-end latency of each message from the moment the sender actor (on the server side) generates the message until the moment the receiver actor would be ready to send it to the remote client.

Figure 3 uses letter-value plots to show the distribution of latencies using different scheduling policies on the AMD and Intel machines. LAS/L has higher throughput and better latency distribution than the other two policies on both AMD and Intel machines. LAS/A has significant lower throughput than both RWS and LAS/L and the latency distribution shows a higher average latency with a long latency tail that goes up to 1000 seconds on both machines. Profiling indicates that the workload is dominated by deque lock contention, similar to PUBSUB. This contention causes variable and high latency numbers, and fairly low throughput, because threads are relatively often blocked on a lock.

Hence, although LAS/A is performing fairly good in simple scenarios and benchmarks, lock contention significantly affects its performance when the application scales and number of actors increases. Therefore, the proposal in [26] does not apply to large-scale actor-based applications. The LAS/L policy, on the other hand, shows stable performance improvements over RWS and reduces the latency.

VI. Conclusion

This paper present a study of the effectiveness of locality-aware schedulers for actor runtime systems. Various characteristics of the actor model and message passing frameworks are investigated that can affect execution performance on NUMA machines. In addition, the applicability of existing work-stealing schedulers is discussed. We use these findings to develop two variants of a novel locality-aware work stealing scheduler (LAS) for the C++ Actor Framework (CAF) that takes into account the distance between cores and NUMA nodes. The performance of LAS/L and LAS/A is compared with CAF's default randomized victim scheduler. Locality-aware work stealing shows comparable or better performance in most cases. However, it is also demonstrated that the effectiveness of locality-aware schedulers depends on the workload. In particular, affinity-unblocking can cause lock contention.

Acknowledgment

This work has been supported by the Natural Sciences an Engineering Research Council of Canada (NSERC) and Huawei Technologies.

References

[1] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, Winter 2007.

[2] Z. Majo and T. R. Gross, "Memory System Performance in a NUMA Multicore Multiprocessor," in *Proc. 4th Ann. Int'l Conf. Systems and Storage*, 2011, pp. 12:1–12:10.

[3] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *Proc. 18th Int'l Conf. Parallel Architectures and Compilation Techniques*, 2009, pp. 261–270.

[4] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Advance Papers of the Conf.*, vol. 3. Stanford Research Inst., 1973, p. 235.

[5] C. Hewitt and H. G. Baker, "Actors and Continuous Functionals," Massachusetts Inst. of Technology, Tech. Rep., 1978.

[6] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[7] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting Actor Programming in C++," *Computer Languages, Systems & Structures*, vol. 45, pp. 105–131, April 2016.

[8] G. Agha, "Concurrent Object-oriented Programming," *Commun. ACM*, vol. 33, no. 9, pp. 125–141, Sep. 1990.

[9] 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. 6th Int'l Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2016, pp. 31–40.

[10] J. Armstrong, R. Virding, C. Wikström, and M. Williams, "Concurrent Programming in ERLANG," 1993.

[11] J. Bonér, "Introducing Akka - Simpler Scalability, Fault-Tolerance, Concurrency & Remoting through Actors," 2010, http://jonasboner.com/introducing-akka.

[12] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil, "Deny Capabilities for Safe, Fast Actors," in *Proc. 5th Int'l Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2015, pp. 1–12.

[13] R. D. Blumofe et al., *Cilk: An Efficient Multithreaded Runtime System*. ACM, 1995, vol. 30, no. 8.

[14] A. Pop and A. Cohen, "OpenStream," *ACM Trans. Architecture and Code Optimization*, vol. 9, no. 4, pp. 1–25, 2013.

[15] OpenMP, "OpenMP Application Program Interface Version 4.0," 2013.

[16] J. Yang and Q. He, "Scheduling Parallel Computations by Work Stealing: A Survey," *Int'l J. Parallel Programming*, pp. 1–25, 2017.

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

[18] Z. Vrba, H. Espeland, P. Halvorsen, and C. Griwodz, "Limits of Work-Stealing Scheduling," in *Job Scheduling Strategies for Parallel Processing*, 2009, pp. 280–299.

[19] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable Work Stealing," in *Proc. Conf. High Performance Computing Networking, Storage and Analysis*, 2009, pp. 53:1–53:11.

[20] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler," in *2010 IEEE Int'l Symp. Parallel Distributed Processing (IPDPS 10)*, 2010, pp. 1–12.

[21] S. Min, C. Iancu, and K. Yelick, "Hierarchical Work Stealing on Manycore Clusters," in *In 5th Conf. Partitioned Global Address Space Programming Models*, 2011.

[22] E. Francesquini, A. Goldman, and J. F. Mhaut, "A NUMA-Aware Runtime Environment for the Actor Model," in *2013 42nd Int'l Conf. on Parallel Processing*, 2013, pp. 250–259.

[23] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP Task Scheduling Strategies for Multicore NUMA Systems," *The Int'l J. High Performance Computing Applications*, vol. 26, no. 2, pp. 110–124, 2012.

[24] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 4:1–4:28, Dec. 2012.

[25] W. Suksompong, C. E. Leiserson, and T. B. Schardl, "On the Efficiency of Localized Work Stealing," *Information Processing Letters*, vol. 116, no. 2, pp. 100–106, 2016.

[26] Z. Vrba, P. Halvorsen, and C. Griwodz, "A Simple Improvement of the Work-stealing Scheduling Algorithm," in *Proc. Int'l Conf. Complex, Intelligent and Software Intensive Systems*, 2010, pp. 925–930.

[27] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The Data Locality of Work Stealing," in *Proc. 12th Ann. ACM Symp. Parallel Algorithms and Architectures*, 2000, pp. 1–12.

[28] Q. Chen, M. Guo, and Z. Huang, "Adaptive Cache Aware Bitier Work-Stealing in Multisocket Multicore Architectures," *IEEE Trans. Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2334–2343, 2013.

[29] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling Task Parallelism on Multi-socket Multicore Systems," in *Proc. 1st Int'l Workshop on Runtime and Operating Systems for Supercomputers*, 2011, pp. 49–56.

[30] L. L. Pilla et al., "A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems," in *Proc. 41st Int'l Conf. Parallel Processing*, 2012, pp. 118–127.

[31] A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen, "Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages," *ACM Trans. Architecture and Code Optimization*, vol. 11, no. 3, pp. 1–25, Aug. 2014.

[32] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach, "Scalable Task Parallelism for NUMA," in *Proc. Int'l Conf. Parallel Architectures and Compilation (PACT 16)*, 2016.

[33] F. Broquedis et. al, "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications," in *2010 18th Euromicro Conf. on Parallel, Distributed and Network-based Processing*. IEEE, 2010, pp. 180–186.

[34] S. Aronis et al., "A Scalability Benchmark Suite for Erlang/OTP," in *Proc. 11th ACM SIGPLAN Workshop on Erlang Workshop*, 2012, pp. 33–42.

[35] S. M. Imam and V. Sarkar, "Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries," in *Proc. 4th Int'l Workshop on Programming Based on Actors Agents & Decentralized Control*, 2014, pp. 67–80.

[36] Riot Games, "Chat Service Architecture: Servers," Sep. 2015, https://engineering.riotgames.com/news/chat-service-architecture-servers.