Actor Scheduling for Multicore Hierarchical Memory Platforms

Emilio Francesquini
Grenoble Informatics Laboratory
University of Grenoble
Institute of Mathematics and Statistics
University of São Paulo
emilio@ime.usp.br

Alfredo Goldman
Institute of Mathematics and Statistics
University of São Paulo
gold@ime.usp.br

Jean-François Méhaut
Grenoble Informatics Laboratory
University of Grenoble
CEA/DRT/LETI,
Grenoble, France
jean-francois.mehaut@imag.fr

Abstract
Erlang applications are present in several mission-critical systems. These systems demand substantial computing resources that are usually provided by multiprocessor and multi-core platforms. Hierarchical memory platforms, or Non-Uniform Memory Access (NUMA) architectures, account for an important share of these platforms. Yet, the research on the suitability of the current virtual machine (VM) for these platforms is quite limited. The current VM assumes a flat memory space, thus not performing as well as it could on these architectures. The NUMA environment presents challenges to the runtime environment in fields varying from memory management to scheduling and load-balancing. In this article we summarize some of the characteristics of an actor based application to, in light of the above, introduce some NUMA-aware improvements to the Erlang VM. This modified VM uses the NUMA characteristics and the application knowledge to take better memory management, scheduling and load-balancing decisions. We show that, when we consider the default Erlang VM as the baseline, the modified VM can achieve performance improvements up to a factor of 2.50 while limiting the slowdown on the worst case by a factor of 1.15.

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

General Terms Experimentation, Measurement, Performance

Keywords multi-core, NUMA, pinning, profiling, actor model, Erlang

1. Introduction
The actor model, base for Erlang’s concurrency primitives, was originally proposed in the context of artificial intelligence [11] but only a few years later it began to be regarded as a possible model for concurrency [2]. This concurrency model has several interesting characteristics such as memory isolation, message exchange and serial processing of messages by each process. These aspects of the model allow for the nonexistence of locks, semaphores or any other synchronization specific tool. Although powerful, this abstraction – willingly – takes the application developer away from the architectural idiosyncrasies of the machine. Thus, the runtime environment (RE) becomes the responsible for the efficient use of the underlying architecture.

Erlang’s implementation of the actor model has some characteristics that differentiate it from other programming models. For instance, with some few exceptions, a process lifespan is usually very short. Processes are created to perform specific tasks and then they are discarded. Moreover, processes are frequently created in far greater quantities than the number of available processing units (PU). The vast number of processes and their independence makes the actor model a good choice to take advantage of the new multi-core machines.

In 2004, as the first multi-core processor for the mainstream market was unveiled, one could already realize the new trend in processor development strategies for the following years [13]. Consequently, performance improvement has become as much of a software problem as it was, until then, an exclusive hardware problem. Nowadays multi-core and many-core processors already are the norm both on professional and personal environments. Many-core processors such as those from Tilera’s Tile-Gx family [28] and Kalray’s MPPA [4], take this paradigm even further, already offering processors with 72 and 256 cores respectively. Even mobile low power consumption processors, such as Samsung’s Exynos 5 Octa [1] and NVIDIA’s Tegra 4 [17], already have more than four cores. Multi-core processors marked the general adoption of shared cache levels. While larger shared cache levels simplify the internal workings of these processors, they may also cause increased cache contention, and unpredictable variations in execution time.

Symmetric multiprocessor (SMP) is the customary architecture used for multiprocessor machines. As its name states, in this architecture each processor has a symmetric access to memory and I/O. This is accomplished using a common bus for every processor of the system. However, as the number of processors increases, this common bus represents a bottleneck to the overall system performance. One possible solution to avoid this bottleneck is the use of hierarchical memory platforms commonly referred to as non-uniform memory access (NUMA) architectures. In a NUMA architecture the machine is divided in nodes. Each NUMA node has its own processors and local memory. NUMA nodes are connected by a bus that provides global memory access to every processor on the

1 In this text we will follow the established Erlang nomenclature for actors, i.e., we will use the term process to refer to each Erlang VM internal process and we will explicitly state if we are referring to an operating system process otherwise.
system. Since the performance of the NUMA interconnect is noticeably inferior to that of the local NUMA node bus, the access to some regions of the memory may take longer than others, hence the name non-uniform memory access.

NUMA architectures have been the preferred choice of hardware developers for machines with a large number (typically more than a few dozen) of cores. This has been in part motivated by the fact that these architectures are able to run applications developed for SMP machines with no modifications. These pieces of software, however, should take the hardware characteristics into consideration, otherwise concurrency for the shared resources may become strong enough to cause important performance degradation [12, 27]. Efficient software utilization of these machines has been a very active field of research [8, 16, 24, 25]. However, there has been little or no published research on the adaptation of the Erlang VM (or any other actor based RE) for these platforms.

The Erlang VM works as an additional layer over the operating system and, as such, it has supplementary information about the behavior of the application. This information can be employed to make better scheduling and load-balancing decisions. Possible strategies include: analysis of the communication graph between the processes to better place actors during runtime, hierarchical work-stealing and load balancing, actor pinning, and hub processes identification and placement.

This document analyzes and describes actor based applications and the Erlang VM. To illustrate our analysis, we present a real actor based application. Our analysis covers the suitability of the current VM to the NUMA platform followed by a set of simple, yet quite efficient, improvements. These modifications, as far as we know, are original and, therefore, they are not being currently used yet quite efficient, improvements. These modifications, as far as we can, however, draw some conclusions about the RE from this simple example. First, the actor RE must be very efficient for the execution of short-lived processes, otherwise applications such as CouchDB would not perform well. For short-lived processes, the decision of the initial placement, i.e., the choice of the PU in which the process is going to be executed, must be fast. Second, the process scheduler must be able to deal with copious amounts of processes and with their creation in bursts. The MapReduce [9] model, used by CouchDB, Riak [22] and many other actor based applications, does exactly that: it creates many short-lived processes in a short period of time. Third, a typical short-lived process is (or will try to be) active for most of its life. Nonetheless, the number of processes in the application typically exceeds the number of available PUs making a time sharing solution a necessary part of the RE.

2.1.1 Process lifespan

Not every process is created equal. Every application has specialized processes to perform distinct kinds of work. It would be a futile exercise to try to list every possible type of process. We can, however, define two major categories of interest and analyze their general properties. In this context we are interested in the short and long-lived processes.

Our illustrative application, CouchDB, creates many short-lived processes. Actually, 99.5% of the processes live less than 1.5s and 88.9% less than 0.1s. The time needed to create a (minimal) process on this same machine is, on average, 1.5μs.

The real proportion between short and long-lived processes is application specific. We can, however, draw some conclusions about the RE from this simple example. First, the actor RE must be very efficient for the execution of short-lived processes, otherwise applications such as CouchDB would not perform well. For short-lived processes, the decision of the initial placement, i.e., the choice of the PU in which the process is going to be executed, must be fast. Second, the process scheduler must be able to deal with copious amounts of processes and with their creation in bursts. The MapReduce [9] model, used by CouchDB, Riak [22] and many other actor based applications, does exactly that: it creates many short-lived processes in a short period of time. Third, a typical short-lived process is (or will try to be) active for most of its life. Nonetheless, the number of processes in the application typically exceeds the number of available PUs making a time sharing solution a necessary part of the RE.

2.1.2 Process Communication Costs

Communication between processes in the VM is done through message exchanges. Erlang’s provides an efficient implementation of this service which is based on shared memory. Shared memory communication costs on NUMA machines are defined not only by the size of the message but also by the location of the sender and receiver processes. In NUMA platforms, communication costs can easily become one of the determining factors of the application performance [8]. Figure 1 shows the performance penalty incurred to send messages of different sizes considering the cost between processes on the same PU, i.e., sharing the level one cache, as the baseline. For smaller messages, the inter-node performance can be more than seven times slower while for bigger messages the performance is about half that of the baseline. For the intra-node case, the performance is about three and two times worse for small and big messages respectively.

Processes that have an important flow of communication between them and are not optimally placed may cause some undesirable effects. In addition to the longer time of communication, these effects might include, for example, contention on the hardware interconnections such as the NUMA links and increased number of cache misses.

In order to show one of these effects, we created a simple artificial application (depicted in Figure 2). This application is intended to demonstrate the impact of a bad process placement on the cache. During the execution, we handpicked communicating processes and deliberately placed them as close as possible (on the
same core) and compared it with the second best placement (distinct core, same socket). This minimal migration caused approximately 1000 times more cache misses than the optimal placement.

2.1.3 Hub Processes

Process usually have a well and pre-defined function when they are created. Some of these functions are naturally more requested than others, making the communication and load pattern not uniform. We call hub processes, those processes that exchange significantly more messages than the average process and that communicate with a wide variety of distinct processes. We define the set of the processes that exchange messages with a hub process as the hub affinity group.

To illustrate these definitions, we have analyzed the communications graph of CouchDB. The graph depicted in Figure 3 is a representation of the communications that happened during an actual execution.

The information about the hubs and their affinity group is available to the VM during the execution of the application. Thus, during the definition of the migrations, the RE load-balancer could take the affinity group of a hub into consideration. The rationale here is to minimize the communication costs between the hub and its affinity group therefore improving the overall system performance.

2.2 The Erlang Virtual Machine

The development of the Erlang language and VM took into consideration several aspects of the actor model such as asynchronous message passing, private mailboxes, dynamic actor creation, and no shared data. In this section we examine in more detail some of the strategies and policies of the Erlang VM.

2.2.1 Process Scheduler

Actor model REs are usually built to use one of two approaches: the thread-based or the event-based approaches. The main difference between the two is that in the former each actor is represented by an operating system process or thread, whereas in the latter each actor is represented by an internal RE data structure. In this case, the RE is responsible for the scheduling of actors and the overall system load-balancing instead of the OS. While this makes the RE more complex, it also makes it more powerful since the RE has the opportunity to perform runtime optimizations that would not have been possible otherwise.

The event-based approach is the choice made by the Erlang VM while, for example, Scala [19] gives the developer the option to choose between them. In the Erlang VM actors are represented as a simple data structure. This explains how a typical Erlang application, that has dozens (sometimes thousands) of processes, can be run efficiently on machines with just a few PUs.

By default, the Erlang VM creates one OS thread for each available PU (Figure 4). These threads are called schedulers. Even though binding schedulers to the available PUs could improve performance by providing a better use of the processor caches, they are not bound by default.

Each scheduler has a run queue that keeps the runnable processes assigned to it. A process is said to be runnable when it is not waiting for messages or any other blocking operation. When scheduled for execution, it will run until its pre-determined share of the processor runs out or it is blocked by some I/O operation. At that point, the process will be preempted by the scheduler and put back on the run queue. The scheduler will then schedule the next process on the run queue for execution.
During the application execution, the sizes of the queues of each scheduler might become very different from each other. Even if the queues were balanced in the beginning of the execution, each process has distinct lifespans. Moreover, when a new process is spawned it is assigned to the same run-queue of its parent and not every process has the same process spawning behavior.

To control this imbalance, the Erlang VM employs two strategies: work-stealing and periodical load-balancing. If a scheduler runs out of processes to execute it will start a migration logic to steal work from the other schedulers. This would be enough to keep all the schedulers busy if there is enough work for all of them. However, work-stealing by itself is not enough to ensure that each process receives a fair share of the available PUs: each queue might have a different size yielding ineffective distinct processor shares. Hence the VM periodically runs a load-balancing routine between the run-queues.

The criteria used to determine which and how many processes will be migrated from which and to which run-queues do not take into consideration the process affinity group or if it is a hub. Only the size of the queues and the position of the process in that queue is considered.

2.2.2 Process Memory Management

Erlang processes share no data. Each one of them has its own heap. This simplifies the implementation of an efficient garbage collector since such a collector does not need to “stop the world”, inasmuch as it only needs to stop the process on which it is working. In fact, many short-lived processes never experience a garbage collection during their lifetimes being completely discarded once they are done increasing the overall system performance.

Since the heap of each process is independent, the exchange of messages is done by copying. That is, every message gets copied from the heap of the sender process to the heap of the receiving process. There is, however, one exception: the binary type. Binaries bigger than 64 bytes are allocated in a shared binary heap and they are sent in messages by reference rather than by value.

Heap allocation is an intrinsic task related to the spawning of a new process. The heap of a process is allocated by the scheduler of the parent process, meaning, that the scheduler responsible for the execution of the parent is also responsible for the allocation and copying of the spawned process parameters. In flat-memory space machines, the location of the allocated memory does not vary, it is always local. On the other hand, on NUMA machines, the operating system can employ several different policies to memory placement. Linux, for example, uses by default a first-touch policy. For the Erlang VM this means the spawned process heap location will be on the node where the scheduler that created it was running. We will call this location the process home node.

It is important to note that home nodes are not definitive. Take for example a process that, for whatever reason, was migrated. During its execution it might need to grow its heap to fit new data. Often it is not possible to allocate additional memory using the same memory address and, in this case, a full heap copy to the new location must be done. If the new scheduler to which the process was migrated is not on the same node as the process’s home node, its home node will be changed and any RE functionality that depends on this information will need to be updated. Moreover, the cost of a simple heap growth operation that would have been proportional to the size of the heap on a flat memory space machine now also depends on the current process location and home node.

It appears that some of the choices taken by the developers of the Erlang VM may suggest that is was not written with the NUMA architecture in mind. This is not, however, specific to the Erlang VM. To the best of our knowledge, no other current actor model (language based or library based) RE takes into consideration the NUMA aspects of the machine. In the following section, we discuss some considerations the RE might employ to become better suited to NUMA platforms.

3. A NUMA-Aware Approach

NUMA platforms present challenges not only to actor model REs but also to any concurrent application. The distinct costs to access different parts of the memory cause a considerable number of problems that, among others, involve process and memory placement, scheduling, load-balancing, and memory migration. We are interested in ways to efficiently exploit these platforms using the currently available VM with few modifications. In order to do that, we analyzed the behavior of some actor based applications. More specifically, we studied their communication graph and their hubs behavior.

We were looking for common patterns in the execution of these applications and the analysis of the communication graphs yielded two main conclusions. First, hub processes usually are responsible for the creation of the majority of the processes that belong to its affinity group (this heuristic will be heavily used in our approach). Second, the communication graph and, therefore, the affinity group of processes, are extremely dynamic. Trying to maintain an online representation of the graph or of the affinity group could bring an important overhead to the RE. We therefore propose a simpler approach based on some hints from the application developer.

Developers often have good insights into the execution characteristics of the application. They can, therefore, hint possible hubs. Hints do not change the functional behavior of applications and the RE could, at its own discretion, completely disregard them. However, the RE can also use them to make better decisions. Our approach works by giving the developer tools to flag the processes he believes are hubs. It can be done during the process spawning, meaning that the developer has, at the moment of creation, some evidence that the spawned process will be a hub. This kind of evidence can also come up during the execution. A later decision probably means that it depends on the evaluation of data that is only available during runtime. For example, processes chosen by on-line election algorithms might become hubs during the application execution, therefore changing their profile after their creation.

Our proposal is based on two main aspects of the RE, the load-balancing policy and the process affinity maintenance. Load-
balancing aims not only at maintaining every available scheduler busy most of the time, but also to ensure that every process gets a fair share of the PUs time. The process affinity maintenance tries to keep processes, and their affinity group, close together so that communication between them is fast. Sometimes these two goals may conflict. For example, maximum process affinity could be achieved if we were to place every process together on the same scheduler, however that would leave the remaining schedulers idle thus minimizing the load balance. We are after good trade-offs, in terms of performance, between these two aspects of the execution.

Different actor model REs face load-balancing in very different ways. Thread-based REs usually delegate the solution to the operating system. A natural solution considering that each actor is an operating system thread. On the other hand, event-based REs solve it themselves, normally using single or multiple run queues. The single-queued version essentially works by employing a thread pool that consumes work from this queue. This version has no need for a separate load-balancing logic. On the other hand, the multiple-queued version has a separate run queue for each thread. In this case, as the behavior of each actor is different, there might be some imbalance between the queues. That is why work-stealing and load-balancing algorithms are employed.

The thread-based model imposes some limits on what one is able to observe and act upon since the load-balancing decisions are taken by the operating system. We will therefore concentrate on the event-based approach. Among the event-based approaches, the single-queued one is the simplest. It works very well in a flat memory space machine with a small number of PUs. However, as the number of execution flows grows larger, the contention to access the common queue increases, thus limiting the scalability of the system [15]. Furthermore, on a NUMA platform, threads will probably be distributed throughout the whole machine. In this scenario, the common queue will distribute the processes execution evenly across the threads. This causes processes to bounce between threads, creating a significant number of cache misses therefore increasing the traffic on the NUMA interconnection. In other words, this solution does not promote soft-affinity. On the other hand, multiple-queued approaches have one queue per thread, thus, as long as the threads are bound to the PUs, soft-affinity is an intrinsic property. However, this approach has to take into consideration the eventual imbalance between the queues. It is at this point that the work-stealing and load-balancing algorithms are put into place. A loaded system tends to have a small number of migrations therefore preserving soft-affinity. When this is not the case, compaction of load algorithms try to avoid migrations by decreasing the number of active schedulers to a minimum. These reasons compel us to believe the multiple-queued event-based is the most appropriate solution for the actor model RE on a NUMA platform.

Our proposal to maintain both the load balance and the process affinity is centered around the RE load-balancing mechanisms. By modifying how the placement and migration of processes are done, we can, using the heuristic described in the beginning of this section, do both things at once. The approach is divided in the following complementing categories.

### Initial Process Placement

There are several possible policies to place a newly spawned process depending on its expected behavior. Proportionally, hubs demand a lot more from the RE than their regular counterparts. They usually are among the biggest spawners in an application. Thus, it makes sense to try to spread the hubs in a way they do not need to compete for resources. On the other hand, regular processes are likely to communicate within their affinity set, so it makes sense to place them close to their hubs. We propose the use of two different initial placement policies, one for hubs and other for regular processes. Hubs should be spread throughout the available schedulers, while regular processes should be placed near their hub/affinity group, on the same NUMA node. The best way to spread hubs will depend on the application behavior. For example, we could privilege communication between hubs by placing them close but not on the same scheduler (compact), or privilege resource independence by placing processes as far as possible (scatter). Both these strategies promote a good initial distribution of hubs among the available cores.

### Hierarchical Load-balancing and Work-Stealing

During the execution of the applications, imbalances are bound to happen even with a good initial placement policy. That is why the RE needs a periodic load-balancer. Moreover, if a run queue becomes empty between the load-balancing rounds, a work-stealing solution might be employed as a temporary lightweight solution to keep the schedulers busy. Both algorithms will migrate processes between run queues, however, to improve the overall performance of the system on a NUMA platform, the way a candidate is chosen for migration matters. The steps we propose to choose a migration candidate are, first, migrate processes back to their home node and, if that is not enough, migrate processes inside the same node. Only if these steps are not enough, consider the remaining processes for inter-node migration. These steps aim at keeping and restoring the proximity between processes in the same affinity group while maintaining them close to their home node and therefore their heap.

Our proposal for a NUMA-aware Erlang VM has several details that would not be possible to implement and test using the original unmodified VM. For that reason, we have introduced several modifications that allowed us to trace and modify various behaviors of the VM. We outline these adaptations in more detail now.

### 3.1 The Modified Erlang VM

We have used Erlang R15B02 code base as the starting point for our modified VM. Most of the changes we have introduced to the code are platform independent. Nevertheless, some of them (mostly those related to NUMA APIs) are Linux specific. However, we believe that they are generic enough to be easily ported to any other NUMA-aware operating system such as FreeBSD and Windows. We highlight the most important changes we have introduced to the VM’s code base in the following sections.

#### 3.1.1 Processes

An important part of our proposal involves the tagging of hub processes. On the VM code level, a process is represented by a struct and the list of the alive processes is kept by the VM as a plain C array. The size of this array determines the maximum (parameterizable) number of processes in the system. When a process is created, an available slot on the array is filled with the pointer to it and when it dies the slot is emptied. The search for an empty slot, although fast, is done using linear search. Unlike this approach, we chose to maintain the list of the hub processes using an intrusive list in the process data structure. Thus, the flag that indicates if a process is a hub, is actually a linked list node. Not only does this choice facilitate the maintenance of the hubs list (when we need to, for example, remove a dying process from the list), but it also allows us to efficiently list all the hubs.

We have introduced some modifications to make it possible for Erlang code to mark processes as hubs. These hints can be given as an extra parameter during the process creation or by setting a flag of a running process. Depending on the chosen VM options, it might
have no effect at all. On the other hand, given the right parameters, the modified VM will pin the execution of hub processes to a specific scheduler and avoid their migration due to load-balancing or work-stealing. This is done using an undocumented feature of the VM that binds the execution of a process to a specific scheduler. Listing 1 shows the code to create a regular and a hub process as well as the code to mark an existing process as a hub.

Listing 1. Erlang interfaces to create and set hub processes

```erlang
% Regular process
Pid1 = spawn_opt(A_Module, A_Function, FunctionArgs, []).
% Hub process
Pid2 = spawn_opt(A_Module, A_Function, FunctionArgs, [hub_process]).
% Marks calling process as a hub
erlang:system_flag(hub_process, true).
```

The work-stealing and load-balancing modifications also required us to keep track of the home node of a process. We define the home node of a process as the NUMA node where its heap is allocated. Since, during a heap increase operation, the scheduler running the process might not be on the same NUMA node of the previous heap allocation, the home node of a process might change throughout the execution of the application. 

3.1.2 Initial Placement

The default policy of the Erlang VM is to place each newly spawned process on the same scheduler of its parent, therefore, expediting its creation specially due to the memory allocation and heap initialization. We have modified the VM and the user can now choose (using the `erlang:system_flag/2` function call with the atoms `scheduler_ip_strategy` and the desired strategy as parameters) in runtime among the following initial placement options:

- **default** Erlang’s default policy, i.e., places processes on the same scheduler of their parent.
- **compact** Favors communication between processes by placing them close together.
- **scatter** Privileges resource independence by placing processes as far as possible from each other.
- **circular** Performs a round-robin algorithm to distribute the spawned processes.
- **random** Chooses a random scheduler.

The initial placement strategy receives as argument the newly spawned process and its parent. To implement a new strategy, it suffices to create a new function (that receives these two parameters and returns the index of the chosen scheduler), and to add a pointer to it in the list of available strategies.

It is also possible to specify that the chosen strategy is only valid for hub processes (`erlang:system_flag(scheduler_hubs_only, Boolean)`). In this case the default strategy will be used for the regular actors. The currently selected initial placement policy can be queried with the `erlang:system_info/1` function call using the same atom identifiers.

3.1.3 Process Migration

Each scheduler on the VM has a sequential identifier. Depending on the command line options passed during the initialization of the VM (using the `+sbt` option), the threads represented by these identifiers will be bound to the available PUs following different possible strategies such as, for example, `unbound` (default), `no_spread`, `thread_spread`, `processor_spread`, `spread` and their NUMA variations. The full list of available scheduler binding parameters can be found on the Erlang documentation.

Whenever a scheduler runs out of work, a work stealing algorithm is initiated. The current strategy of the Erlang VM consists of first trying to steal processes from an inactive scheduler. If unsuccessful, the algorithm tries to steal from the remaining ones. The candidate schedulers are evaluated on ascending identifier order starting from the stealer identifier plus one, wrapping the search to the first scheduler when it reaches the end of the list. In other words, if the VM has `n` schedulers, the scheduler `i` will try to steal processes from the schedulers in the following order `i + 1, i + 2, ..., n - 1, 0, 1, ... , i - 1`.

Compaction of load is one of the strategies that, when applied with work stealing and load balancing, can improve performance on systems that are not fully loaded. It is enabled by default and can be used to improve process affinity or resource independence. It works by suspending schedulers when the charge of the system is not high enough. With less active schedulers for the same amount of processes, schedulers tend to run out of work less often therefore minimizing the number of calls to the work stealing algorithm. This by itself minimizes bouncing of processes between schedulers. Moreover, as the suspended schedulers are always those with the highest identifiers, given that appropriate scheduler bindings are used, compaction of load can be used to improve process affinity on a NUMA architecture by, for example, migrating processes to a smaller set of PUs on the same NUMA node or to spread them throughout the available NUMA nodes to promote resource independence. This is however a partial solution. It has no effect on loaded systems, for in this case the schedulers are never suspended. Additionally, it offers a very limited view of the NUMA architecture. Usually NUMA nodes are not organized in a linear fashion, limiting the impact of this strategy.

For this reason, we have altered the work stealing code to take into consideration both the architecture of the machine as well as the home node of each process. Similar to our previous modifications, the behavior of the work stealing algorithm can be defined or queried using the `erlang:system_flag/2` and `erlang:system_info/1` function calls passing as argument the atom `scheduler_ws_strategy`. The available behaviors are `default`, `disabled`, and `numa`. The disabled strategy completely disables the work-stealing algorithm and, since it decreases the overall system performance, it is useful only for debugging purposes. Similar to the default algorithm, the numa strategy works by trying to steal from suspended schedulers and only then, if not successful, from the remaining schedulers. From the suspended schedulers it will first try to bring processes home, i.e., it will try to find a process for which the home node is the same of the stealing scheduler. If no process fulfills this criterion, it will try to steal any process. If no process is found during this step, the algorithm will try to steal from the active schedulers. Analogous to the previous step, it will first try to bring processes home and only then consider the remaining processes. Another important difference from the default strategy is that, instead of scanning the list of schedulers sequentially, it will search the schedulers on the same node (therefore not changing the process home node) and only then consider the remaining schedulers.

Roughly speaking\(^1\) the current load-balancer only takes into account the total number of processes and the number of active schedulers. It determines the desired number of processes per sched-

\(^1\)A full description of the load-balancer is out of the scope of this paper. However, a more comprehensive explanation of the load-balancer inner-workings (including how this algorithm deals with suspended schedulers and compaction of load) is provided by Zhang [29].
uler and establishes migration paths. Each scheduler periodically checks these migration paths and pulls/pushes processes from/to other schedulers. While this is a fundamental aspect of the VM to ensure soft real-time properties, no information about the topology of the machine is ever considered. This is an intricate part of the VM code and we chose, for the time being, to modify it as little as possible. Our modification does not concern the generation of the migration paths but actually it deals with the order in which they are scanned. This ensures that we will first try to migrate processes inside the same NUMA node and only then between NUMA nodes. Section 7 discusses further what could be done to improve the load-balancer for NUMA machines.

To ensure the performance of these modifications, each scheduler maintains a list of the foreign-processes, i.e., processes which home node and the scheduler NUMA node are not the same. The implementation of the foreign-processes list is also done using an intrusive list inside the process data structure. In addition to that, each run-queue keeps pointers, indexed by the home nodes, to the intrusive list inside the process data structure. In addition to that, the implementation of the foreign-processes list is also done using an intrusive list inside the process data structure. In addition to that, each run-queue keeps pointers, indexed by the home nodes, to the intrusive list inside the process data structure.

3.1.4 Tracing

To evaluate and debug our modifications we have extended the available Dtrace probes. We included, for example, probes to trace VM activities related to process migration, work-stealing and load-balancing. We have also modified some of the existing probes to include additional information as, for example, those related to process spawning (parent process, initial placement, parent process placement and hub hints) and message sending (location of the sending and receiving processes).

4. Evaluation Methodology

4.1 Evaluated Platforms

In order to evaluate the proposed modifications, we have used two distinct NUMA machines. Table 1 summarizes their general characteristics. In both platforms the tests were run with Hyper-Threading disabled. Simplified architectural views of these platforms are depicted in Figures 5 and 6.

4.2 Erlang VM

We have used Erlang OTP R15B02 as the basis for our tests. The default VM represents our modified version of the VM running with all our modifications disabled. Throughout the text, we will refer to the unmodified VM as vanilla. The modified VM was compiled using the same default parameters of the vanilla VM on the platforms described in Section 4.1. Section 3.1 summarizes the modifications we introduced to the Erlang VM in order to do our experiments.

4.3 Benchmarks

To evaluate the performance of our modified VM, we have used the BenchErl benchmark suite [3]. BenchErl suite has benchmarks to evaluate several different aspects of the Erlang VM. CPU-bound and Erlang language specific APIs benchmarks (such as those that test ETS tables and erlang:now/0) were removed since they are irrelevant to the aspects we want to test, i.e., those where the communication and the placement of the processes have an important role.

We have slightly modified the benchmarks code. Our modification was limited to the addition of the hint needed to inform the VM about the hubs. We briefly describe the chosen benchmarks below.

**bang** Many-to-one message passing. This benchmark creates a set of processes that send at the same time a number of messages to only one receiver. This receiver is clearly a communication hub and his affinity set is the whole set of processes.

---

**Table 1. Platform Specifications**

<table>
<thead>
<tr>
<th>NUMA 32</th>
<th>Altix UV 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMA Nodes</td>
<td>4</td>
</tr>
<tr>
<td>Cores</td>
<td>32</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.27 GHz</td>
</tr>
<tr>
<td>Total RAM</td>
<td>64 GiB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>24 MiB</td>
</tr>
<tr>
<td>NUMA Factor$^*$</td>
<td>1.2 to 3.6</td>
</tr>
<tr>
<td>Linux Kernel</td>
<td>3.5,7</td>
</tr>
<tr>
<td>GCC</td>
<td>4.7,2</td>
</tr>
</tbody>
</table>

---

$^*$NUMA factor is the ratio between remote latency and local latency.
big This benchmark creates a number of processes that send, all at the same time, messages to every other process in the system. The benchmark evaluates how long the RE takes to deliver every sent message. The communication graph is a full graph. There is no communication hub that stands out and the affinity group of each process is composed of every other process in execution.

orbit_int It is an implementation of a distributed hash table (DHT). In this DHT, each bucket is a process. To add something to the DHT, a process sends a message with the data to be inserted to the appropriate bucket. Upon the reception of this message, the bucket/process might need to process it before storing it. This can take some time thus it is done in parallel by the creation of multiple worker processes. When it is completed, additional data that must be stored on the DHT might have been generated. Their generators, the worker processes, then send these data back to their master that in turn forwards them to the appropriate buckets for storage. The benchmark measures how long the RE takes to insert a specific set of data into the DHT, including the time needed to process it and insert any additionally generated data. This benchmark has clearly defined communication hubs: the buckets. These processes are involved in most communications and perform the role of a master process that spawns several workers.

ehb It is an Erlang implementation of the Hackbench [30] stress test for schedulers. It works creating several groups of communicating processes. Each of these groups has one coordinator that spawns a set senders and receivers processes (its affinity group) and receives a message back from these processes when they are done. The coordinators were hinted as hubs.

serialmsg In an extreme case of communications bottleneck, this benchmark has only one communication hub. This process acts as a message dispatcher for every other process on the system. The benchmark works by creating two sets of processes, the senders and the receivers. The communication between these two sets of processes is done, exclusively, through this dispatcher. The affinity group of this process is the whole set of processes.

timer_wheel This benchmark spawns a set of processes that exchange ping and pong messages. Both the ping and pong messages are sent and received by every process. It is quite similar to the big benchmark, however the reception of the pong messages can be limited by a timeout. Like that benchmark, the communication graph can be a full graph and there is no clear communication hub and the affinity group of each process is the set of the remaining processes.

5. Experimental Results

The vanilla Erlang VM has some optional parameters that set execution policies capable of improving the performance on NUMA architectures. To test the performance of our approach, we have taken as a baseline the benchmark execution performance without the use of any optional VM policies. We will refer to this configuration as original. However, for the sake of a fair comparison, we present next to our results, the performance obtained by the best tuning of policies using only the options present on the the vanilla VM. We will refer to this configuration simply as default. Similarly, the performance results that made use of the policies we have proposed in this paper will be referred to as modified. Table 2 describes the policy options available on the original and modified versions of the VM.

We have extensively tested the modified VM on the NUMA 32 platform, the results for this platform are shown in Section 5.1. We also show our preliminary results for the Altix UV 2000 platform on Section 5.2, a machine with a higher NUMA node count in which neither the vanilla nor the modified VMs were able to efficiently execute.

5.1 NUMA 32

Figure 7 depicts the normalized execution time of the chosen benchmarks, with respect to the original configuration. We show results for both the short and intermediate data input sizes. In this machine, we were able to to improve the performance in benchmarks: bang, ehb, orbit_int, and timer_wheel. Table 3 shows the full list of the obtained speedsups.

![Figure 7. Normalized execution time of the benchmarks for two different data input sizes on the NUMA 32 platform.](image-url)

Let us start by analyzing the performance of the orbit_int and ehb benchmarks. These benchmarks have clearly defined communication hubs: the buckets and the coordinators. These processes are involved in most communications and perform the role of a
The two benchmarks we could not improve the performance, big and serialmsg, deserve more attention. These are peculiar benchmarks that specifically test the RE against extreme communication situations. In the first, there is no communication hub that stands out and the affinity group of each process is composed of every other process in execution. In the latter there is only one communication hub. These two benchmarks present the RE with situations that do not fit well on the assumptions of our proposal. Examples of these kind of situations include those where every process of a system is a hub, where no process is a hub, or when there is just one hub. Our proposal assumes that the application will have a few communication hubs and that we will be able to spread their affinity groups throughout the NUMA nodes of the machine. When the application has only one communication hub and its affinity group is the whole set of processes, our approach ends up introducing overheads that we are not able to compensate. For these specific situations, a simpler approach, such as that of the original VM, might be better since it does not impose additional overheads.

The timer_wheel benchmark is an interesting case by itself. Every policy change we attempted to improve its performance had the opposite effect. Actually, the performance gains we have shown for this benchmark were obtained with every, original and proposed, alternative policy disabled. The reported speedup comes from the fact that, even with every policy disabled, our modified virtual machine strives not to migrate hub processes. The considerable decrease on the number of process migrations (see Figure 8) is responsible for the reduction of 53% (small) and 47% (intermediate) on the reported execution time.

**Performance Impacting Factors**

There are several factors behind the reported improvements in performance. First, we have a better initial placement of processes. A bad initial process distribution means that some schedulers will become overloaded, and thus processes on these schedulers will have a higher level of processor sharing. Since many processes are short lived, this wait for the processor might make a considerable difference in their lifespans. Another more important reason, however, is that a better initial process distribution means a reduced number of load-balancing migrations. A process migration, by itself, has a cost that is non negligible. Moreover, the execution of the process on a NUMA node different of his home node imposes additional overheads. Figure 8 shows the average number of migrations using the default and the modified configurations. Some of the benchmarks that had a significant increase in performance such as orbit_int and timer_wheel have also shown a considerable reduction on the average number of migrations.

Some factors influence some benchmarks much more than others. This explains why some of them performed substantially better despite the fact that the number of migrations was kept practically constant. Some benchmarks, such as ehb are much more susceptible to alterations on the initial placement than to alterations on scheduler bindings. On the other hand, benchmarks like bang are more influenced by the scheduler bindings than by any other policy. Table 4 shows the average reduction of the execution time for each of the evaluated policies using the short data set.

An intriguing fact is that the compaction of load policy almost did not change the overall execution time of the benchmarks. This was, however, to be expected. Every benchmark we executed creates many more processes than the available number of schedulers. This renders the compaction of load essentially inactive for most of the time.

**Vanilla VM Comparison**

The performance results we presented, although measured using the default behavior of the Erlang VM, were assessed using the modified VM. This was done not only due to the limitations of the original VM (for example, there is no support to trace process migrations), but also because we are interested in comparing the impact of each distinct policy only. We did not fine tune the modified VM code. Thus, the comparison between the modified VM performance and that of the heavily optimized vanilla VM would defeat the purpose of our experiment. We have, however, estimated the overhead our modified code imposes to the execution of the benchmarks. Our measurements show an overhead ranging from 2% to 26%. Such an overhead range allows us to, in some cases, employ the modified VM and still obtain significant performance gains even without the code optimizations. Figure 9 depicts the execution of the benchmarks using two distinct workloads. We show for each benchmark the performance of the best tuning for the vanilla and modified VMs. Execution times were normalized by the vanilla VM execution time with no optional policy parameters.
Figure 9. Normalized execution time of the BenchErl benchmarks using the best tuning for the vanilla and modified VMs on the NUMA 32 platform. Execution times were normalized by the vanilla VM execution time with no optional parameters. Benchmarks were executed, at least 30 times each, using intermediate and short configurations.

5.2 Altix UV 2000

In this section we present the preliminary results we obtained on the Altix UV 2000 platform. In this machine, we were only able to improve the performance of two benchmarks. Figure 10 depicts the performance comparison between the default and modified configurations and Table 5 shows the full list of the achieved speedups.

Table 5. Altix UV 2000- Obtained Speedups

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bang</td>
<td>0.87</td>
<td>0.99</td>
</tr>
<tr>
<td>big</td>
<td>1.23</td>
<td>0.93</td>
</tr>
<tr>
<td>ehb</td>
<td>1.08</td>
<td>1.02</td>
</tr>
<tr>
<td>orbit_int</td>
<td>1.30</td>
<td>1.07</td>
</tr>
<tr>
<td>serialmsg</td>
<td>0.98</td>
<td>0.95</td>
</tr>
</tbody>
</table>

Table 6. Benchmarks Execution Time

<table>
<thead>
<tr>
<th></th>
<th>NUMA 32</th>
<th>Altix UV 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>bang</td>
<td>0.89s</td>
<td>110.48s</td>
</tr>
<tr>
<td>big</td>
<td>0.43s</td>
<td>3.23s</td>
</tr>
<tr>
<td>ehb</td>
<td>1.01s</td>
<td>11.38s</td>
</tr>
<tr>
<td>orbit_int</td>
<td>1.14s</td>
<td>8.59s</td>
</tr>
<tr>
<td>serialmsg</td>
<td>2.25s</td>
<td>412.27s</td>
</tr>
<tr>
<td>timer_wheel</td>
<td>1.92s</td>
<td>117.62s</td>
</tr>
</tbody>
</table>

Table 7. Measured Slowdown

<table>
<thead>
<tr>
<th></th>
<th>NUMA 32</th>
<th>Altix UV 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>bang</td>
<td>16.6</td>
<td>4.5</td>
</tr>
<tr>
<td>serialmsg</td>
<td>1.3</td>
<td>1.6</td>
</tr>
</tbody>
</table>

This platform, however, requires further analysis. The execution times for both the vanilla and the modified VMs is much longer than we expected. Table 6 shows the execution time in seconds for the short configuration of the benchmarks using the vanilla VM on NUMA 32 and Altix UV 2000 platforms.

The comparison of the execution times makes it clear that something is slowing down the execution of the benchmarks on this platform. The Altix UV 2000 platform has six times more processors and a far better NUMA interconnection than the NUMA 32 platform and yet we are not able to completely explain the performance loss. However, we believe the communication bottleneck created by the NUMA architecture of this machine is probably one of the main reasons for the observed performance difference.

The NUMA Bottleneck

As in any platform, it is essential that applications make good use of the resources at hand. In a NUMA platform, this kind of care is even more important. bang and serialmsg are two radical examples that explicitly show how extreme the performance losses can become if the application is poorly written. In both cases, only one process concentrates the communications of the whole system.

To measure these slowdowns, we have used the vanilla VM with no optional parameters using the short configuration of the benchmarks. The VM was therefore run using one scheduler per available PU, i.e., 32 schedulers on NUMA 32 and 192 on Altix UV 2000. We took the unrestricted execution of the VM and compared it to the execution of the VM limited to one NUMA node using the same number of threads. In other words, on NUMA 32 we ran the VM on 32 PUs with one thread per PU (4 nodes) and also using 4 threads per PU (1 node). Similarly, for Altix UV 2000 using one thread per PU (192 PUs, 24 nodes) and using 192 threads in just one node (24 threads per PU). The idea behind this setting is to obtain a rough estimate of how big an influence the NUMA interconnection has on these ill-behaved applications. If it is not a bottleneck, we would see performance degradations since the restricted execution would force the operating system to share the PU, essentially giving each thread smaller processor shares. However, our results listed in Table 7 show exactly the opposite: the restricted execution has a speedup of up to 16.6 when compared to the performance of the regular execution.

As we pass from only one NUMA node to the complete machine, we stop using the local bus and local cache-coherence protocols and begin to employ the NUMA interconnects to exchange messages between processes. Additionally, in our case, every process on the system sends messages to only one process and, therefore, only to one NUMA node. This creates an important bottleneck.

5 We used the numact1 Linux utility to restrict the access to PUs and memory of the VM to just one NUMA node.
on the NUMA interconnect slowing down the execution even further. While this kind of implementation is, by itself, problematic in both SMP and NUMA architectures, the effects caused by the increased cost in communication on NUMA platforms are much more noticeable. Moreover, even with fast NUMA interconnects such as those on the Altix UV 2000 platform, 23 nodes sending messages to only one node can easily jam the communication channels. As it might be expected, in both platforms the CPU utilization is quite low during the execution of the benchmarks and, therefore, this could be used as a possible indication to identify this kind of problem.

6. Related Work

The research involving scheduling of actors to better use multi-core machines has mostly been about the development of SMP capable REs [10, 15] and the analysis and optimization of their performances [18, 29]. These works focused on the evaluation of the available implementation choices such as single or multiple run-queues, lock-free message passing, software transactional memory, etc. There has also been some research for the development of alternative actor based concurrency frameworks [26]. However, to the best of our knowledge, there is no published research that considers the hierarchical topology of the memory (intrinsinc to NUMA platforms) to implement a NUMA-Aware actor RE.

The Release Project [5] aims to create a high-level paradigm for reliable large-scale server software. Among others, one of their goals is to evolve the Erlang VM so that it can effectively work on large scale multi-core systems. The BenchErl suite of benchmarks we used has been created by this project. Until now, however, its main focus has been on the distributed aspects of the RE.

On the other hand, the research community has shown a strong interest for non-actor concurrent NUMA-Aware REs. Rashiti [23] et al. show how a better match between the communication and physical topologies on MPI applications can bring considerable gains in communication performance. Charm++ on NUMA machines face many of the same problems of actor-based REs and it has been shown that [20] NUMA awareness might bring improvements on the overall system performance.

Improvements on the REs are not the only possibilities currently in exploration. Kernel level solutions such as AutoNUMA and NumaSched [25] try to improve the performance of the processes transparently doing better memory allocation and scheduling distributions. In particular, NumaSched also has the notion of a process home node. A process will allocate memory preferentially from its home node. The scheduler will restrict the execution of a process to its home node unless load-balancing dictates otherwise. In this case, a migration may end up changing the home node of the process. This will be followed by a lazy memory page migration. NumaSched, on the other hand, employs a different approach. For each process the kernel maintains the last NUMA nodes of the memory pages it has recently accessed. Similarly, for each page the kernel maintains the last NUMA node that accessed it. Based on these statistics, the kernel decides if (and where) a process or a memory page needs to be migrated. Unfortunately, this kind of approach has limited efficiency on REs (for the actor model or not) that do not have a direct link between each internal flow of execution and an operational system thread or process. Furthermore, the actor RE has additional higher level information, opaque to the Kernel, that can be used to make better scheduling and memory allocation decisions.

Our proposal involves a programming methodology (actor model) and the scheduling and load-balancing mechanisms necessary to make it more efficient on NUMA multi-core platforms. We have applied our proposal to a specific RE, nonetheless, our approach is not exclusive to the Erlang programming language or RE. We strongly believe that the techniques applied to this particular environment are also applicable to other actor REs. While our proposal aims at being transparent to the user, we still give him the possibility to hint hubs. These hints are not obligatory and do not alter the functional behavior of the application.

7. Future Work

For performance reasons, different strategies for memory allocation are employed by the Erlang VM depending on the kind of data at hand. For example, process heaps, binaries and ETS data have their own custom allocators. Among these, we can highlight process heap allocation as one of the most important. The process heap is allocated when the process is created and also when there is a change in its size. When a process is created, its heap is allocated on the same NUMA node of the scheduler running the spawning process. A better heap allocator could take into consideration the chosen initial placement of the process and allocate the memory on the corresponding NUMA node promoting a better memory utilization.

The migration-paths generation makes the simplifying assumption that the interconnection of the NUMA nodes is a full graph. The use of the real interconnection topology and performance might further improve the results we obtained on some machines. This information could be used, for example, to minimize both the number of process migrations between NUMA nodes and the number of processes away from their home nodes while at the same time minimizing the communication costs.

Our assumption that hub processes are, for most of the time, responsible for the creation of the majority of the processes that belong to its affinity group is very strong and may not be true for every application. Moreover, home nodes can be changed during the execution. Our approach uses the current location and the actors home node as an indication of the actor proximity to its hub and, therefore, affinity group. By removing the need to use such heuristic, we expect to improve the effectiveness of our approach. Moreover, we have limited our tests to the synthetic benchmarks of the BenchErl suite. A more comprehensive performance evaluation with real applications is one of our priorities.

I/O performance on NUMA platforms is also variable depending on the NUMA node. Each I/O device, as for example a hard-disc or network-adapter, is connected to one specific NUMA node. A process that does I/O operations on these machines should be scheduled to run specifically on these nodes to avoid traffic on the NUMA interconnection and therefore improve performance. Additionally, further research on the reasons of the weak Altix UV 2000 platform performance is clearly required.

All the modifications we presented are on a branch of the original Erlang OTP Git repository. We intend to optimize our code and adapt it to the Erlang OTP standards so that we can submit a patch for inclusion of these features into the official VM. These changes include making the code compatible with other operating systems such as FreeBSD.

8. Conclusion

In this paper we investigated the behavior of actor-based applications and the impact of these characteristics on NUMA machines. In light of this characterization, we proposed a set of new optimizations for the Erlang VM on these platforms. These optimizations intend to minimize the undesirable performance degradation effects caused by the NUMA platform on the RE and, consequently, on the applications.

We modified the VM to implement the proposed optimizations and evaluated them using two distinct NUMA platforms. Additionally, we have made it simple to add new strategies for work-
stealing, load-balancing and initial placement of processes. Among other modifications, we can highlight the addition of hierarchical work-stealing capability as well as the scatter and compact strategies for the initial placement of processes.

To assess the impact of our optimizations, we evaluated the performance of the modified Erlang VM using standard benchmarks. Our experiments show that some simple NUMA-Aware optimization policies can bring significant performance improvements (up to a factor of 2.5) to the RE and therefore to the applications. We have also shown that poorly coded applications, that might display acceptable performances on SMP architectures, might experience severe performance losses on these hierarchical memory platforms.

Acknowledgment
The authors would like to thank CAPES for funding this research under project CAPES/Cofecub 660/10.

References


