Improving the Performance of Actor Model Runtime Environments on Multicore and Manycore Platforms

Emilio Francesquini  
Grenoble Informatics Laboratory  
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

The actor model is present in many systems that demand substantial computing resources which are often provided by multicore and multiprocessor platforms such as non-uniform memory access architectures (NUMA) and manycore processors. Yet, no mainstream actor model runtime environment (RE) currently in use takes into account the hierarchical memory characteristics of these platforms. These REs essentially assume a flat-memory space therefore not performing as well as they could. In this paper we present our proposal to improve the performance of these systems. Using knowledge about the general characteristics of actor-based applications and the underlying platform, we propose techniques spanning from memory management to scheduling and load-balancing. Based on previous work, we present our design guidelines for the RE adaptation to the Kalray MPPA-256 manycore processor.

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

Keywords actor model, multicore, manycore, NUMA

1. Introduction

Symmetric multiprocessor (SMP) is the customary architecture used for multiprocessor machines. As its name states, in this architecture each processor has symmetric access to memory and I/O. This is accomplished using a common bus for every processor of the system. Since the first multicore processor hit the mainstream market in 2004, performance improvements have become as much of a software problem as it was, until then, an exclusive hardware problem. Nowadays multi-core and many-core processors are the norm both on professional and personal environments. Some processors such as those from the Tilera Tile-Gx [21] and Kalray MPPA [2] families, take this paradigm even further, offering processors with 72 and 256 cores respectively.

However, as the number of processors (and cores per processor) increases, this common bus might represent a bottleneck to the overall system performance. One common solution is the use of hierarchical memory platforms usually referred to as non-uniform memory access (NUMA) architectures. These machines are divided into nodes that have their own processors and local memory. Nodes are connected by a bus that provides global memory access to every processor on the system. Since the performance of the NUMA interconnect is noticeably inferior to that of the local NUMA node bus, accesses to some regions of the memory take longer than others, hence its name. Another common approach, typical of many-core processors, consists on the use of a network-on-chip (NoC). In these processors, two communicating processing units (PUs) might need to use the NoC to exchange messages.

Although powerful, the actor model and the platform abstraction it provides – willingly – take the application developer away from the architectural idiosyncrasies of the machine. Thus, the actor model runtime environment (RE) becomes the responsible for the efficient use of the underlying architecture. However, to the best of our knowledge, no current mainstream actor RE takes into account these characteristics, essentially assuming a flat-memory space platform.

NUMA architectures have been the usual choice of hardware developers for machines with more than a few dozen processors. This was motivated, in part, by the fact that these architectures are transparently able to run applications developed for SMP machines. These applications, however, should take the hardware characteristics into consideration, otherwise concurrency for the shared resources may become strong enough to cause important performance degradation. It has been shown that an efficient utilization of these machines by the actor REs can result in performance improvements up to a factor of 2.50 while limiting the slowdown on the worst case by a factor of 1.09 [7, 8].

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

In this document we describe a work in progress aiming at the actor REs performance for multicore and manycore platforms. It summarizes our previous works on the adaptation and evaluation of actor REs for NUMA machines and introduces our design guidelines for the adaptation and development of these REs for the MPPA-256 manycore platform.

This paper is organized as follows. Section 2 describes the hardware platforms, actor model application characteristics, and how current REs employ it. Then Section 3 details our proposal for the improvement of the REs on non-SMP platforms. Next, we compare our approach to related works in Section 4. Finally, we discuss future works and conclude in Section 5.
2. Background

In this section we briefly describe the target hardware architectures and the typical actor RE inner workings. We then analyze some general characteristics and behavior of actor-based applications.

2.1 Multicore Multiprocessor and Manycore Platforms

The demand for higher processor performance has made chipmakers include into their designs solutions that are a combination of brute-force and innovation. The increase of processors cache size, instruction-level parallelism and working frequency have been for the last decades their main tools to accomplish this mission. However, these approaches seem to have reached a point in which they, by themselves, are not enough to ensure the steep curve of performance improvement predicted by Moore’s Law and expected from the consumers [10]. An exponential increase in power consumption related to a linear increase in the clock frequency [6] and a higher complexity to design new processors changed their course of development to include an increasing number of independent cores, therefore letting application parallelism take over.

NUMA architectures appeared as a solution to the scalability of multiprocessor SMP platforms. NUMA machines are normally composed of several multicore processors divided into nodes with their own memory. Although there is a global memory space, access to regions of memory on other nodes is typically more expensive. We call NUMA factor the ratio between remote latency and local latency for memory access in these machines. Some machines might display NUMA factors beyond 10 [8], making a correct placement of processes and memory essential to obtain good performances [4].

Similar to NUMA platforms, on a manycore the communication between cores is accomplished through the use of a NoC. Often the NoC interconnections are not a complete graph, therefore letting application parallelism take over. On the other hand, Kalray’s MPPA-256 maycore (see Section 3.3) groups a total of 256 cores into clusters of 16 cores. Each cluster has local memory accessible only to its cores. Cores in the same cluster communicate using a common bus (much like an SMP platform) while cores in different clusters communicate through a processor-wide NoC built using a 2D torus topology. Both NUMA and MPPA-256 manycore platforms display similar characteristics, i.e., NUMA nodes group processors and memory while on the MPPA-256 platform, cores are grouped into clusters with their own local memory. Fast access to local memory and an increased variable cost to access data on a different node/cluster present challenges not only to the actor model REs but also to any concurrent application. These challenges involve, process and memory placement, scheduling, load-balancing, and memory migration. We are interested in ways to efficiently exploit these platforms using currently available REs with few modifications. In order to do that, we analyzed the behavior of some actor-based applications and REs which we present in the following sections.

2.2 Actor Model Programming

The following sections present some aspects of the current actor REs and applications with a special attention to the Erlang RE. This specific RE was used as the base for our prototype (outlined in Section 3). Whenever necessary we point out the differences between this specific RE and other implementations. However, it is our belief that the techniques we employed for this specific implementation of the actor model are generic enough to be used by any actor RE.

2.2.1 Actor Model Applications

Some aspects of the actor model can be directly used to improve the performance of the available REs. The knowledge of the actors behavior, their communication graph, the communication costs of the target machine and the relationship between the actors are all examples of knowledge available to the RE that can be used to improve its own scheduler and load balancer decisions.

Actor lifespan Every application has specialized actors to perform distinct kinds of work. It would be impractical to try to list every type of actor. 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 actors. It has been shown that in a typical application most actors have a very short lifespan [7]. For that reason the actor RE must be very efficient for the execution of short-lived actors, otherwise these applications would not perform well. For short-lived actors, the decision of the initial actor placement, i.e., the choice of the core in which the actor is going to be executed, must be fast. Moreover, the actor scheduler must be able to deal with copious amounts of actors and with their creation in bursts. The MapReduce model, commonly used by actor-based applications such as Riak and CouchDB [17] does exactly that: it creates many short-lived actors in a short period of time. Additionally, the number of alive actors in the application typically exceeds the number of available PUs making a time sharing solution a necessary part of the RE.

Actor Communication Costs In principle, every communication on the actor model is based on message passing. How it is actually done depends not only on the RE implementation but also on the underlying platform. On SMP and NUMA platforms it is safe to assume that an efficient implementation will be done using shared memory. On the other hand, on the MPPA-256 platform, the RE will forcibly employ shared memory for intra-cluster and the NoC for inter-cluster communications.

Contrary to SMP machines, communication costs on NUMA and manycore machines are defined not only by the size of the message but also by the sender and receiver locations. In these platforms, communication costs can easily become one of the determining factors of the application performance [5].

For small and big messages, inter-node message exchanges on NUMA platforms can be respectively more than seven and two times slower than a local message exchange [7]. Moreover, actors that have an intense flow of communication between them and are not optimally placed may cause an increased number of cache misses in addition to the contention on the hardware interconnections (such as the NUMA links and the NoC).

2.2.2 Hub Actors

Actors usually have a well and pre-defined function when they are created. Some functions are naturally more requested than others, making the communication and load pattern among actors non-uniform. We call hubs those actors that exchange significantly more messages than the average actor and communicate with a wide variety of distinct actors. We define the set of the actors that exchange messages with a hub actor as the hub’s affinity group [7]. Figure 1 illustrates these definitions.
Information about the hubs and their affinity group is available to the actor RE 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 intent is to minimize the communication costs between the hub and its affinity group therefore improving the overall system performance.

2.2.2 Actor Runtime Environments

This section describes in general lines how the current actor REs work with special interest in two of their fundamental aspects: scheduling and memory management.

Actor Scheduler Actor model REs are usually built to use one of two approaches: the thread-based or the event-based approaches. In the former each actor is represented by an OS thread or process, while 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 [1] while, for example, Scala [13] lets the developer choose between them. Additionally, since actors are represented as a simple data structure, event-based REs are able to efficiently run an application with hundreds, sometimes thousands, of actors, even on machines with just a few PUs. Normally this is not the case for thread-based REs.

Event-based REs mostly work by creating one OS thread for each available PU. This pool of threads is the responsible for the execution of the actors. We call scheduler each one of these threads. Occasionally these threads are pinned to each available PU, essentially creating a direct relation between schedulers and PUs. In this approach, ready-to-run active actors are kept in a queue. An actor is said to be runnable when it is not waiting for messages or any other blocking operation. There are some implementations that use a single queue for all the actors while others use one queue per scheduler. On the multiple-queued version, 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. The initial placement and load balancing policies of each actor becomes, therefore, an important runtime aspect of the RE.

Actor Memory Management Actors share no data and, although the exact details are implementation dependent, this is normally how the REs deal with it. In Erlang, for example, each actor has its own heap [8]. This makes it easy to implement an efficient garbage collector since such a collector does not need to “stop the world”, inasmuch as it only needs to stop the actor on which it is working. Other REs, as for example Scala, do not have a dedicated specific VM so a private heap is harder to implement. Actually, in Erlang, since actors are short lived, many can be directly discarded never experiencing a garbage collection during their lifetimes.

As the heap of each actor is independent, the exchange of messages is usually done by copying, i.e., every message gets copied from the heap of the sender actor to the heap of the receiving actor. Depending on the RE, the exchange of messages might be made by reference rather than by value. In this case, the RE ends up assuming the responsibility to ensure that the data will not be changed/garbage collected until the receiving actor has used it.

Heap allocation is an intrinsic task related to the spawning of a new actor. The heap of an actor is usually allocated by the scheduler (or thread for a thread-based RE) of the parent actor, meaning, that the scheduler responsible for the execution of the parent is also responsible for the allocation and copying of the spawned actor’s parameters. This, as we will see in Section 3, is an important aspect for an efficient memory management.

3. A Hierarchical Approach

Actor-based systems are used in a wide range of applications varying from embedded systems to social networking sites. Many of these large-scale services have an ever increasing need for computing resources which are typically provided by NUMA, and more recently, manycore platforms.

Current REs are highly optimized to run on SMP machines. They are able to shield the application developer from the hardware idiosyncrasies and at the same time provide good performance. However, some characteristics of the NUMA and manycore platforms call into question many of these optimization decisions, specially on a NUMA platform which is capable of running unmodified SMP code. Among these characteristics one is patent: communication. Communication between PUs on an SMP platform is simple and fast. The system’s common bus simplifies cache coherence protocols and provides the framework for an efficient low-latency communication. On NUMA and manycore architectures, cache-coherence (if available) must be guaranteed by the use of the NUMA interconnections or the NoC.

Based on the knowledge of these platforms and on the analysis of the actor-based applications, our proposal for improving the performance of these systems can be divided in two main aspects: affinity group maintenance and memory management.

Affinity Groups Maintenance We were looking for common patterns in the execution of actor-based applications and our analysis of the communication graphs yielded two main conclusions. First, hub actors usually are responsible for the creation of the majority of the actors that belong to its affinity group (this heuristic will be heavily used in our approach). Second, the communication graph and consequently the affinity group of actors, are extremely dynamic. Trying to maintain an on-line 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. Hence they can provide hints about possible hubs. Hints do not change the functional behavior of applications and the RE can, at its own discretion, completely disregard them. However, the RE can also use them to help it make better decisions. Our approach works by giving the developer tools to flag the actors believed to be hubs. It can be done during the actor spawning, meaning that the developer has, at the moment of an actor creation, some evidence that the actor 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, actors chosen by online election algorithms might become hubs during the application execution, thus changing their behavior after their creation.

Memory Management On SMP platforms, the location of the allocated memory does not vary, it is always local. On the other hand, on NUMA machines, the OS can employ several different policies to memory placement. Linux, for instance, uses by default a first-touch policy. The same is true for the MPPA-256 platform, i.e., the heap will be allocated on the cluster of the spawning scheduler. We will call this location the actor’s home node.

Load-balancing mechanisms are the responsible for actor migration between schedulers in the same RE. That is why these mechanisms are central to our approach. We detail in the next section how, using these mechanisms, we can keep actor’s affinities and perform a better memory management.

3.1 Load-balancing

Load-balancing aims not only at maintaining every available PU busy most of the time, but also to ensure that every actor gets a fair share of the PUs time. To keep actor affinities, the load balancer
tries to keep actors, and their affinity group, close together so that communication between them is fast. Sometimes these two goals may conflict. For example, the maximum actor affinity would be to place every actor 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.

We believe the event-based model with multiple queues is the most appropriate choice for these hierarchical architectures, not only because it does not impose limits on what can be done but also because it provides soft-affinity and eliminates some scalability problems [7]. Furthermore, a solution based on a single queue would be inefficient and of complex implementation on a manycore platform like MPPA-256 on which there is no global shared memory or cache-coherence. 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.

By applying the heuristic described in the beginning of this section, we can modify the actor placement and migration algorithms to keep actors affinities. This approach is divided in the following complementing categories.

**Initial Actor Placement** Proportionally, hubs demand more resources 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 actors are likely to communicate within their affinity set, so it is natural to place them close to their hubs. We propose the use of two different initial placement policies, one for hubs and other for regular actors. Hubs should be spread throughout the available PUs, while regular actors should be placed near their hub/affinity group, on the same node/cluster. The best way to spread hubs will depend on the application behavior, e.g., we can privilege communication by placing hubs close but not on the same PU/Node (compact), or privilege resource independence by placing actors as far as possible (scatter).

**Hierarchical Load-balancing and Work-Stealing** 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 solution to keep the PUs busy. Both algorithms will migrate actors between run queues, however, to improve the performance of the system, the way a candidate is chosen for migration matters. Current REs migrate actors mainly based on the size of the queues. We propose that the topological distance between the queues, more specifically, the distance between the PUs to which the schedulers owners of the involved queues are bound, be taken into consideration. That is, every migration would be first done between queues that are near and only then between queues that are farther.

In the next sections, we describe how these ideas were integrated into a real NUMA-aware RE and provide our design guidelines for an efficient RE for the MPPA-256 platform.

### 3.2 NUMA Approach

We chose a real actor RE as the basis for our modifications. Our choice, the Erlang RE, was based on the fact that it is one of the most prominent actor-based languages today and because it has its own custom made VM. In this section, we briefly describe these modifications and outline the most important conclusions. A full analysis of the modifications and results is available in [8].

One of the central points of the RE modifications is the possibility to mark actors as hubs. We introduced interfaces that allow the application to mark an actor during its creation or at any other point in time. During the spawning of a new actor, the RE must decide where this actor will be placed. By default the Erlang RE places each newly spawned actor on the same scheduler of the parent actor. We modified this behavior and added the following initial placement options: compact, scatter, circular and random. It is also possible to choose different strategies for hubs and regular processes.

We also modified the work-stealing and load-balancing algorithms to take into account the topological distance between the PUs. The modified work-stealing has two phases. First it will try to bring actors home, that is, bring actors which home node is the same of the stealing scheduler but are on a remote node. Only if no actor in this condition is found, it begins the second phase. During this phase it will try to steal actors from cores in the same processor, then on the same NUMA node and only then from the remaining NUMA nodes. During the first phase, NUMA nodes are scanned in decreasing order of distance and in the second phase in increasing order. The rationale behind it is that by bringing home the most distant actors we have a bigger impact in performance. The second phase does the same, i.e., minimizes the migration impact by stealing first from the nearest actors. The load-balancing algorithm was modified to follow the same strategy. These steps aim at keeping and restoring the proximity between actors in the same affinity group while maintaining them close to their home node and therefore their heap.

#### 3.2.1 Experimental Results

We evaluated our modifications on a NUMA platform with 32 cores running at 2.27GHz and a total of 64 GB of RAM equally divided among its four nodes. Although in this machine the NUMA links form a complete graph, it has a NUMA factor ranging from 1.2 to 3.6. For our tests we used Linux Kernel 3.3.7, GCC 4.7.2 and Erlang OTP R15B02. The modified RE was tested against the Bench Erl benchmark suite [18].

The original RE has some parameters that can be set to improve the performance on NUMA platforms. We compare our results using two baselines, Default where no optional parameters are set and Best Tuning in which we use the best configuration found using only the parameters available on the unmodified RE. Table 1 presents the obtained speedups for two different instances of each benchmark: short and intermediate.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Short</th>
<th>Intermediate</th>
<th>Short</th>
<th>Intermediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>serialmsg</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>orbint</td>
<td>1.27</td>
<td>1.14</td>
<td>1.27</td>
<td>1.12</td>
</tr>
<tr>
<td>timer_wheel</td>
<td>2.13</td>
<td>1.89</td>
<td>2.13</td>
<td>1.88</td>
</tr>
</tbody>
</table>

The two cases in which we could not improve performance (big and serialmsg) are peculiar benchmarks that specifically test the RE against extreme communication situations [7]. These benchmarks present the RE with situations that do not fit the assumptions of our proposal. Examples of these kind of situations include those where every actor of a system is a hub, where no actor is a hub, or when there is just one hub. Our proposal assumes the application will have a few hubs and that we will able to spread their affinity groups throughout the NUMA nodes. When the application has only one communication hub and its affinity group is the whole set of actors, our approach ends up introducing overheads that we are not able to compensate. In these cases, a simpler and lighter approach, such as that of the original RE, might be better suited.

### 3.3 Manycore Approach

Some manycore processors, such as Tilera’s TILE64Pro processor, are able to transparently run a single OS that controls the entire platform. As far as applications are concerned, they run under the
Despite of the similarities between NUMA and MPPA-256 to think about this platform using some of the abstractions we used do it is not as simple as referencing local memory, this allows us
communication between clusters and I/O subsystems is done exclusively cache, static memory and external DDR access. There is no cache between PEs. Each I/O subsystem relies on 4 RMs with a shared D-memory of 2MB, which enables a high bandwidth communication between either external buses (e.g. PCIe) or SDRAM. For this reason, cated to run user threads whereas RMs execute kernel routines and RMs and PEs implement the same Very Long Instruction Word that communicate through data and control NoCs [2]. Although are distributed across 16 compute clusters and 4 I/O subsystems user cores (Processing Elements - PEs) and 32 system cores (Re-
Figure 2, is a single-chip manycore processor that integrates 256 provide the illusion of an SMP machine. This platform, depicted in
architectural aspects exists. In this document we concentrate on its characteristics of the MPPA-256 platform.

3.3.1 The MPPA-256 Platform
Unlike Tilera’s processors, Kalray’s MPPA-256 processor does not provide the illusion of an SMP machine. This platform, depicted in Figure 2, is a single-chip manycore processor that integrates 256 user cores (Processing Elements - PEs) and 32 system cores (Resource Managers - RM) in 28nm CMOS technology. These cores are distributed across 16 compute clusters and 4 I/O subsystems that communicate through data and control NoCs [2]. Although RMs and PEs implement the same Very Long Instruction Word (VLIW) architecture, they have different purposes: PEs are dedicated to run user threads whereas RMs execute kernel routines and services of NoC interfaces. Operations executed by RMs vary from task and communication management to I/O data exchanges between either external buses (e.g. PCIe) or SDRAM. For this reason, RMs have privileged connections to NoC interfaces. Both PEs and RMs feature private 2-way associative instruction and data caches. Each compute cluster features 16 PEs, 1 RM and a local shared memory of 2MB, which enables a high bandwidth communication between PEs. Each I/O subsystem relies on 4 RMs with a shared D-cache, static memory and external DDR access. There is no cache coherence between the cores. Contrary to the RMs available on compute clusters, the RMs of the I/O subsystem can also run user code. Moreover, each cluster and I/O subsystem executes its own instance of the OS. Since there is no global address space, communication between clusters and I/O subsystems is done exclusively through the NoC. The NoC has special functions that allow remote I/O operations on remote clusters. Although the code necessary to do it is not as simple as referencing local memory, this allows us to think about this platform using some of the abstractions we used for the NUMA platforms.

3.3.2 Scheduling
Despite of the similarities between NUMA and MPPA-256 architectures, the absence of cache coherence and a global memory space might be one of the biggest challenges to the creation of an efficient RE for this architecture. In this platform, POSIX Threads features such as locks and condition variables issue full memory barrier instructions, limiting the performance of the system. MPPA-256 platform also offers special low-level system calls that, at the expense of performance, allow direct I/O operations on the cluster’s memory, essentially bypassing the cache, as an alternative. The actor RE is responsible for shielding the application developer from all these hardware idiosyncrasies, however choosing the best set of tools to code an efficient communication between the threads of the RE might demand considerable efforts.

The OS running in each cluster has a hard limit of 16 threads per cluster, i.e., one thread per PE or 256 per processor. A custom made actor model RE for this platform has, therefore, a hard choice to make. Either it uses a thread-based approach and limits the total number of actors on the system to 256 or uses an event-based approach. The thread-based approach has the advantage of being lightweight: it does not need extra code or data structures to deal with the scheduling and eventual load-balancing of actors (a non-negligible advantage when we consider that the OS consumes about 500KB of memory and therefore leaves only about 1.5MB of memory for the 16 user threads). As actor-based applications have (and are encouraged to have) a large number of actors, it is our opinion that the event-based approach is the most appropriate choice. A single-queued event-based system would impose severe limitations to the RE, mostly by the excessive use of the NoC. Using one queue per scheduler is not the best solution as well since it could create an unacceptable memory and load-balancing overhead. We thus propose a mixed strategy: To reduce the memory overhead on each cluster, we will use one single queue per cluster, totaling 16 queues in the whole system.

This single queue per cluster simplifies initial placement polici-
ies. Such a strategy places regular actors on the same cluster of the spawning actor. This expedites allocation and initialization of heaps for regular actors, as it will be completely made within the cluster, without the need for communications (crucial for short-lived ac-
tors). For hub actors, this strategy promotes a good initial division of resources spreading them evenly among the clusters.

However, the choice of an event-based solution with multiple-queues forces the RE to deal with eventual imbalances in the system. The proposed hybrid schema allows only for inter-cluster imbalances. Thus the problem is how to devise an efficient inter-cluster load-balancing strategy.

3.3.3 Load-balancing
Contrary to NUMA platforms, it is not acceptable, from a performance point of view, to run an actor in a cluster other than its home. Any migration therefore implies heap relocation. Taking into consideration that this is an expensive operation, the thresholds used to determine if migrations are required might need to be empirically adjusted. Otherwise, migrations can be done following the same proposal we presented for NUMA architectures, with both work-stealing and load-balancing searching other clusters queues for actors in an increasing order of distance. Additionally, in this architecture, it might be needed to keep track not only of the home cluster of an actor but also of its hub. This way, an improved version of the migration routines could increase the performance of the system by providing better actor affinities.

Furthermore, since there is no global memory to deliver a message for example, the RE needs an alternative and efficient way of tracking actor locations. Some systems use the spawning location of an object as a proxy. Whenever there is a migration, the migrating actor signals its original cluster with the new location. The OS running in each cluster has a hard limit of 16 threads per cluster, i.e., one thread per PE or 256 per processor. A custom made actor model RE for this platform has, therefore, a hard choice to make. Either it uses a thread-based approach and limits the total number of actors on the system to 256 or uses an event-based approach. The thread-based approach has the advantage of being lightweight: it does not need extra code or data structures to deal with the scheduling and eventual load-balancing of actors (a non-negligible advantage when we consider that the OS consumes about 500KB of memory and therefore leaves only about 1.5MB of memory for the 16 user threads). As actor-based applications have (and are encouraged to have) a large number of actors, it is our opinion that the event-based approach is the most appropriate choice. A single-queued event-based system would impose severe limitations to the RE, mostly by the excessive use of the NoC. Using one queue per scheduler is not the best solution as well since it could create an unacceptable memory and load-balancing overhead. We thus propose a mixed strategy: To reduce the memory overhead on each cluster, we will use one single queue per cluster, totaling 16 queues in the whole system.

This single queue per cluster simplifies initial placement polici-
ies. Such a strategy places regular actors on the same cluster of the spawning actor. This expedites allocation and initialization of heaps for regular actors, as it will be completely made within the cluster, without the need for communications (crucial for short-lived ac-
tors). For hub actors, this strategy promotes a good initial division of resources spreading them evenly among the clusters.

However, the choice of an event-based solution with multiple-queues forces the RE to deal with eventual imbalances in the system. The proposed hybrid schema allows only for inter-cluster imbalances. Thus the problem is how to devise an efficient inter-cluster load-balancing strategy.

3.3.3 Load-balancing
Contrary to NUMA platforms, it is not acceptable, from a performance point of view, to run an actor in a cluster other than its home. Any migration therefore implies heap relocation. Taking into consideration that this is an expensive operation, the thresholds used to determine if migrations are required might need to be empirically adjusted. Otherwise, migrations can be done following the same proposal we presented for NUMA architectures, with both work-stealing and load-balancing searching other clusters queues for actors in an increasing order of distance. Additionally, in this architecture, it might be needed to keep track not only of the home cluster of an actor but also of its hub. This way, an improved version of the migration routines could increase the performance of the system by providing better actor affinities.

Furthermore, since there is no global memory to deliver a message for example, the RE needs an alternative and efficient way of tracking actor locations. Some systems use the spawning location of an object as a proxy. Whenever there is a migration, the migrating actor signals its original cluster with the new location. The proxy then updates the list which contains the location of all the immigrant actors. When a message is received, if the actor is on the list of immigrants the message is redirected, the sender is notified of the new location of the receiving actor, and the sender updates the actors last known location so the next communication is faster. If another migration happens in the meantime and a message cannot be delivered to the actor’s last known location, the system falls back to the spawning cluster to look for the updated receiver’s last known location. This solution demands a minor modification to the RE, i.e., the reference to an actor would also include information about its spawning cluster and last known location. In our case, however, memory in each cluster is quite limited. If each cluster kept a list of all the actors it spawned, this list would probably consume an unacceptable amount of memory. Fortunately RMs on the I/O subsystems, which are also capable of running user code,
have access to external DDR memory that is orders of magnitude larger than the memory available to each cluster. So our proposal’s only difference is how the list of immigrant processes is kept. The list in each cluster will work as limited-size cache to the full list kept by the nearest I/O subsystem. Whenever a request for update or message is received by the proxy, the cache list is searched. If found, the message is redirected, if not the cache is updated after a request to the nearest I/O subsystem. We envision the use of an LRU eviction policy for this cache, however empirical evaluations of alternative policies should be conducted.

4. Related Work

The research involving scheduling of actors to better use multi-core machines has mostly been about the development of SMP capable REs [9, 12] and the analysis and optimization of their performances [15, 22]. There has also been some research for the development of alternative actor-based concurrency frameworks [20]. However, to the best of our knowledge, there is no published research that considers the hierarchical topology of the memory to implement an improved version of the actor RE.

On the other hand, the research community has shown a strong interest for non-actor platform aware concurrent REs. Rashidi [14] 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 REs and it has been shown that [11] NUMA awareness might bring improvements on the overall system performance. Aubry et al. [16] demonstrate a complete dataflow based RE for the MPPA-256 architecture.

Kernel level solutions such as AutoNUMA and NumaSched [19] 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. 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 and manycore platforms. We strongly believe that the techniques we propose are applicable to any actor RE. While our proposal aims at being transparent to the user, it still allows hubs to be hinted. These hints are not obligatory and do not alter the functional behavior of the application.

5. Conclusion and Future Work

In this paper we investigated the behavior of actor-based applications and the impact of these characteristics have on NUMA machines. We outlined a set of new optimizations intended to mitigate the undesired performance effects caused by the NUMA platform. In light of the positive performance results of these optimizations, we drafted the guidelines for the development of an efficient platform aware actor RE for the MPPA-256 manycore. We detailed the trade-offs between the available implementation choices and the hardware limitations that will need to be dealt with. However, since the MPPA-256 platform works with a POSIX compatible OS, we believe that the current actor REs can be used as a starting point for the creation of a custom platform aware actor-model RE.

As future works we intend to continue with the modifications on the Erlang RE to make it capable of running on the MPPA-256 platform. First, we will need to replace many low-level routines for their MPPA-256 equivalent, then implement the modifications proposed in this paper. We are confident that it is possible to obtain good performance using this platform even if some hard memory-related compromises must be made.

Acknowledgments

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

References