THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

préparée dans le cadre d'une cotutelle entre l'*Université de Grenoble* et *l'Universidade de São Paulo*

Spécialité : **Mathématiques-informatique**

Arrêté ministériel : le 6 janvier 2005 -7 août 2006

Présentée par

Emilio de Camargo Francesquini

Thèse dirigée par **Jean-François MÉHAUT**
codirigée par **Alfredo GOLDMAN VEL LEJBMAN**

préparée au sein des **Laboratoire d'Informatique de Grenoble et Departamento de Ciência da Computação** dans l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique** et l'**Instituto de Matemática e Estatística da Universidade de São Paulo**

Dealing with actor runtime environments on hierarchical shared memory multi-core platforms

Thèse soutenue publiquement le **16 mai 2014**,
devant le jury composé de :

Dealing with actor runtime environments on hierarchical shared memory multi-core platforms

Emilio de Camargo Francesquini

Thesis presented
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
to
obtain the title
of
Doutor em Ciências

Program: Computer Science
Advisor: Prof. Dr. Alfredo Goldman vel Lejbman
Co-advisor: Prof. Dr. Jean-François Méhaut

During the development of this work the author
received financial support from CAPES/COFECUB

São Paulo, May 16, 2014

Dealing with actor runtime environments on hierarchical shared memory multi-core platforms

For Patrícia, my beloved wife.
Your kindness and caring support made it all possible.

Acknowledgements

"Solutions nearly always come from the direction you least expect, which means there's no point trying to look in that direction because it won't be coming from there."
— Douglas Adams, *The Salmon of Doubt*, 2002.

A few years ago I set out on my personal project to become a PhD. Now, with the maturity that only comes with time, I realize that I had only a slight idea of what that really meant. Fortunately, I was not alone in this endeavor. During this journey I had the support of many awesome people that, in one way or another, helped me face the personal and professional barriers brought forth by a task of this magnitude. Without their support this project would not have been made possible.

Firstly, I would like to thank my advisors Alfredo Goldman vel Lejbman and Jean-François Méhaut. The innumerable discussions we had both during my stay in Brazil as well as in France showed me that sometimes it is necessary to take two steps back before we can take one step forward. With them I learned that solutions come naturally (sometimes unexpectedly) to those who are looking for them. I also would like to thank them for providing all the support I needed to develop and publish the research contained in this thesis as well as giving me the opportunity to get in touch with several international researchers.

Secondly, I would like to thank my laboratory colleagues in Brazil (Nelson Lago, Yanik Ngoko, Marcos *Marquito* Amaris, Thiago Furtado, Paulo Moura, Leonardo Leite, Felipe Besson, Guilherme Nogueira, Wesley Seidel and Carlos Eduardo Santos) and in France (Vinicius Pinheiro, Cristian Sanabria, Augustin Degomme, Luka Stanisic, Brice Videau, Gabriel Duarte, Christophe Laferrière, Rodrigue Chakode and Thomas Nguélé). Your company and the great discussions we had made this period of my life memorable.

Finally, I would like to thank all my family. Specially my wife, Patricia. You stood there for me when I needed you the most. I would also like to thank my parents, Sueli and Fernando, and my siblings Fernanda and Felipe. Thanks for believing in me during all my life. I love you all.

Abstract

The actor model is present in several mission-critical systems, such as those supporting WhatsApp and Facebook Chat. These systems serve thousands of clients simultaneously, therefore demanding substantial computing resources usually provided by multi-processor and multi-core platforms. Non-Uniform Memory Access (NUMA) architectures account for an important share of these platforms. Yet, research on the suitability of the current actor runtime environments for these machines is very limited. Current runtime environments, in general, assume a flat memory space, thus not performing as well as they could. In this thesis we study the challenges hierarchical shared memory multi-core platforms present to actor runtime environments. In particular, we investigate aspects related to memory management, scheduling, and load-balancing.

In this document, we analyze and characterize actor based applications to, in light of the above, propose improvements to actor runtime environments. This analysis highlighted the existence of peculiar communication structures. We argue that the comprehension of these structures and the knowledge about the underlying hardware architecture can be used in tandem to improve application performance. As a proof of concept, we implemented our proposal using a real actor runtime environment, the Erlang Virtual Machine (VM). Concurrency in Erlang is based on the actor model and the language has a consistent syntax for actor handling. Our modifications to the Erlang VM significantly improved the performance of some applications thanks to better informed decisions on scheduling and on load-balancing.

Resumo

O modelo de programação baseado em atores é frequentemente utilizado para o desenvolvimento de grandes aplicações e sistemas. Podemos citar como exemplo o serviço de bate-papo do Facebook ou ainda o WhatsApp. Estes sistemas dão suporte a milhares de usuários conectados simultaneamente levando em conta estritas restrições de desempenho e interatividade. Tais sistemas normalmente são amparados por infraestruturas de *hardware* com processadores de múltiplos núcleos. Normalmente, máquinas deste porte são baseadas em uma estrutura de memória compartilhada hierarquicamente (NUMA - *Non-Uniform Memory Access*). Nossa análise dos atuais ambientes de execução para atores e a pesquisa na literatura mostram que poucos estudos sobre a adequação deste ambientes a essas plataformas hierárquicas foram conduzidos. Estes ambientes de execução normalmente assumem que o espaço de memória é uniforme o que pode causar sérios problemas de desempenho. Nesta tese nós estudamos os desafios enfrentados por um ambiente de execução para atores quando da sua execução nestas plataformas. Estudamos particularmente os problemas de gerenciamento de memória, de escalonamento e de balanceamento de carga.

Neste documento nós também analisamos e caracterizamos as aplicações baseadas no modelo de atores. Tal análise nos permitiu evidenciar o fato de que a execução de *benchmarks* e aplicações criam estruturas de comunicação peculiares entre os atores. Tais peculiaridades podem, então, ser utilizadas pelos ambientes de execução para otimizar o seu desempenho. A avaliação dos grafos de comunicação e a implementação da prova de conceito foram feitas utilizando um ambiente de execução real, a máquina virtual da linguagem Erlang. A linguagem Erlang utiliza o modelo de atores para concorrência com uma sintaxe clara e consistente. As modificações que nós efetuamos nesta máquina virtual permitiram uma melhora significativa no desempenho de certas aplicações através de uma melhor afinidade de comunicação entre os atores. O escalonamento e o balanceamento de carga também foram melhorados graças à utilização do conhecimento sobre o comportamento da aplicação e sobre a plataforma de *hardware*.

Résumé

Le modèle de programmation à base d'acteurs a été intensivement utilisé pour le développement de grandes applications et systèmes. On peut citer comme exemples la fonction chat de Facebook ou bien encore WhatsApp. Ces systèmes peuvent avoir plusieurs milliers d'utilisateurs connectés simultanément avec des contraintes fortes de performance et d'interactivité. Ces systèmes s'appuient sur des infrastructures informatiques basées sur des processeurs multi-cœurs. Ces infrastructures disposent en général d'un espace mémoire partagé et hiérarchique NUMA (Non-Uniform Memory Access). Notre analyse de l'état de l'art montre que peu d'études ont été menées sur l'adéquation des environnements d'exécution à base d'acteurs avec des plates-formes à mémoire hiérarchique. Ces environnements d'exécution font en général l'hypothèse que l'espace de mémoire est complètement plat, ce qui pose ensuite de sérieux problèmes de performance. Dans cette thèse, nous étudions les défis posés par les plates-formes multi-cœurs à mémoire hiérarchiques pour des environnements à base d'acteurs. Nous étudions plus particulièrement les problèmes de gestion mémoire, d'ordonnancement et d'équilibrage de charge.

Dans la première partie de la thèse, nous avons analysé et caractérisé les applications basées sur le modèle d'acteurs. Cette analyse a permis de mettre en évidence le fait que les exécutions des applications et benchmarks faisaient ressortir des structures de communication avec des propriétés particulières que les environnements d'exécution se doivent de prendre en compte pour optimiser les performances. La prise en compte du graphe de communication et la mise en œuvre ont été effectuées dans un environnement d'exécution réel, la machine virtuelle (VM) du langage de programmation Erlang. Le langage de programmation Erlang s'appuie sur le modèle d'acteurs avec une syntaxe claire et cohérente pour la gestion des acteurs. Les modifications que nous avons intégrées à la machine virtuelle Erlang permettent d'améliorer significativement les performances grâce à une meilleure prise en compte de l'affinité entre des acteurs qui interagissent beaucoup. L'ordonnancement et la régulation de charge des applications sont également améliorés grâce à une meilleure connaissance de l'application et de la topologie de la plate-forme.

Contents

List of Figures

List of Algorithms

List of Tables

List of Abbreviations

API	Application Programming Interface
BIF	Built In Function
CUDA	Compute Unified Device Architecture
DES	Discrete Event Simulation
DHT	Distributed Hash Table
GPU	Graphics Processing Unit
HPC	High-Performance Computing
hwloc	Portable Hardware Locality
LLC	Last Level Cache
MPI	Message Passing Interface
NoC	Network-on-Chip
NUMA	Non-Uniform Memory Access
OS	Operating System
PAPI	Performance Application Programming Interface
PU	Processing Unit
RE	Runtime Environment
SMP	Symmetric Multiprocessing
SPMD	Single Program, Multiple Data
UMA	Uniform Memory Access
VM	Virtual Machine

CHAPTER 1
Introduction

DEMAND for ever-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. An exponential increase in power consumption related to a linear increase in the clock frequency [BBS+00] and a higher complexity to design new processors are examples of the reasons that made the pursuit of ever-higher performance processors, as it was being done, practically unfeasible.

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 [Lar09]. With the advent of multi-core processors, hardware designers were able to maintain the expected performance improvements and, at the same time, keep the design of the processors in a manageable level of complexity. Consequently, performance improvement has become as much of a software problem as it was, until then, an exclusive hardware problem. Although this new paradigm may face some resistance [Sut05, 1][1], multi-core and many-core processors already are the norm both on professional and personal environments. Even mobile low power consumption processors, such as Samsung's Exynos 5 Octa [2] and NVIDIA's Tegra 4 [NVI13], already have more than four cores while some many-core processors such as Kalray's MPPA-256 have up to 256 cores [ABB+13, DML+13].

While the offer of multi-core processors might be abundant, easy and simple solutions for the development of parallel applications are not. Traditional tools such as mutexes and monitors have always been a source of trouble (such as race conditions, deadlocks, starvation, and non-determinism) for the developers. Several possible solutions have been proposed to try to solve the sort of problems these developers have to face when developing such an application. Most of them aim at facilitating the efficient use of the hardware while at the same time shielding the developer from its idiosyncrasies. Most of the current solutions can be classified in the following categories:

- **Low-level APIs** These are typically low-level Operating System (OS) calls that create new processes or threads and give access to some hardware instructions like check-and-set, fetch-and-add, and their variants. In this category of solutions, the application developer is the responsible for all the synchronization between the distinct execution lines. In order to accomplish it, one usually uses mutexes, semaphores or monitors that are made available by the OS API. One of the most known representatives of this category is the POSIX Threads API [But97].

- **Operating System Services** Contrary to the low-level API, the solutions in this category are high-level services offered by the OS. Parallax OS [MS11] as well as Apple's Mac OS X GCD [3], also available for OpenBSD [4], are examples in this category. In particular, GCD's solution consists of several queues of execution, each one with a different priority, that are made available for the application. The application can, in turn, queue jobs to be executed and be notified upon their completion via a callback. If the application tasks can be

[1]In this document we distinguish citations of scientific publications from online resources. To that end, two citation key formats are used. Author initials and year for scientific publications and sequential numbering for online resources.

divided in several small jobs to be queued and executed by the OS, this model results in a good use of the machine's parallel capabilities.

► **Middlewares, Frameworks and Programming Languages** This category's solutions can be very distinct in what concerns their approaches to parallel execution. Two major examples of middlewares/frameworks are OpenMP and Message Passing Interface (MPI) [Qui04]. Although they might seem to be mutually exclusive, these two particular solutions are frequently used together, for example, in a cluster composed of many multi-core machines: OpenMP for local parallelization and MPI for distribution between the machines.

Programming language approaches strive to hide the details of the machine (and sometimes of the OS) from the application developer. Some examples are Charm++ [KK93] and Cilk [BJK+95], that can be seen as parallel extensions to the C/C++ language. There are also actor model based languages like Erlang [Arm10] and Scala [OAC+04], and transactional memory based languages such as Clojure [Hal09]. Not only do these solutions offer a high-level view of the distinct lines of execution but they also provide a programming methodology, *i.e.*, they influence the application architecture. Applications written in these languages usually are able to seamlessly take advantage of newer machines (even those with many additional processing units). In this case, the execution Runtime Environment (RE) is the responsible for the distribution of work between the available processing units, even if they are not all local.

One of the main disadvantages of the low-level approaches is that the application developer has to know beforehand the topology of the network and/or the architecture of the machine in which the application is going to be executed, otherwise performance penalties might ensue. If platform information is not available beforehand, application developers end up having to write intricate underlying runtime supports to adapt the application behavior to the idiosyncrasies of the actual machine and OS. It is our opinion that, unless strictly necessary, the application developer should only worry about the details concerning the application and not about the hardware or OS supporting its execution. For this reason in this work we will focus on the *Middlewares/Frameworks and Programming Languages* category of solutions.

Most mainstream solutions in this category already have some kind of optimization that take into consideration the hardware platform. We will discuss them further in Chapter 6. However, there is little, or no work at all, directed to the efficient execution of actor model REs on large multi-core platforms.

The actor model is present in several mission-critical systems, such as those supporting WhatsApp [5, 6] with 430 million users (with more than one million simultaneous connections) and Facebook Chat [7, 8] with more than one billion users (the actual number of simultaneous connections has not been publicly disclosed). These systems serve thousands of clients simultaneously, therefore demanding substantial computing resources usually provided by multiprocessor and multi-core platforms. Non-Uniform Memory Access (NUMA) architectures (*cf.* Chapter 2) account for an important share of these platforms. Yet, little or no research has been done on the suitability of the current actor REs for these machines. Current REs assume a flat memory space, thus not performing as well as they could. NUMA platforms present challenges to actor model REs in fields varying from memory management to scheduling and load-balancing. In this document we analyze and characterize actor based applications to, in light of the above, propose improvements to these REs.

As a proof of concept, we have applied our ideas in a real actor RE, the Erlang Virtual Machine (VM). Erlang has a dedicated open-source VM which uses the event based approach for the actor RE implementation. The language itself was created based on the original actor model with some minor adaptations and, as such, it has a clean and consistent syntax for actor handling. Moreover, we believe that Erlang is a good choice of language because, since it is close to the original actor model description, the conclusions we draw from this environment can be almost directly applied to other actor-based languages such as Salsa [VA01]. This modified VM takes advantage of the NUMA characteristics and the application knowledge to take better memory management, scheduling and load-balancing decisions.

1.1 Objectives and Thesis Contributions

The actor model was originally proposed in the context of artificial intelligence [HBS73] and only a few years later it also began to be regarded as a possible model for concurrency [Agh86]. In this model, every distinct execution flow is considered an actor. An actor can spawn other actors and there is no shared data between them. The only way to observe or alter the state of an actor is to send or receive a message to and from it. To communicate, each actor has a private mailbox. The messages can be sent to any other actor that, in turn, processes them asynchronously, at its convenience and not necessarily in the order of reception. If an actor runs out of messages to process, it may suspend its execution and keep this status until a timeout is reached or a new message is delivered to its mailbox. The delivery of messages to the actor's mailbox is independent of its state, *i.e.*, even if the actor is busy the delivery of new messages is not blocked.

The memory isolation, the exchange of messages and the serial processing of the messages by each actor allows for the nonexistence of locks, semaphores or any other synchronization specific tool. Actual synchronization between actors is achieved through the exchange of messages. Although powerful, this abstraction – willingly – takes the application developer away from the architectural idiosyncrasies of the machine. Thus, the actor RE becomes the responsible for an efficient use of the underlying architecture.

Actors have some characteristics that differentiate them from other programming models. For instance, with some few exceptions, an actor lifespan is usually very short. Actors are created to perform very specific tasks and then they are discarded. Moreover, actors are frequently created in far greater quantities than the number of available PUs. The reason behind it is two-fold. First, actors are, in general, mostly inactive since for most of their lives they are just waiting for messages. Second, actors keep the state of a system; there is no shared memory, so in order to access data that must be shared among many actors, for example, one needs to create an actor to hold that value. The vast number of actors and their independence makes the actor model a good choice to take advantage of the new multi-core machines.

NUMA architectures have been a tendency on machines with a large number of cores. This has been motivated, in part, by the fact that these architectures are able to run regular applications developed for flat memory space architectures with no modifications. However, if their specific hardware characteristics are not taken into consideration, concurrency for the shared resources might cause important performance degradation [KCS04, TASS07]. As we will show in Chapter 6, efficient utilization methods for NUMA platforms exist but, to the best of our knowledge, there is almost no published research dealing with the adaptation of actor model REs for these platforms.

In order to tackle this problem, we decided to divide our approach in two distinct fronts. First we will investigate how actor applications behave and, with that knowledge, we will adapt and evaluate currently existing REs.

► **Understanding Actor Applications** It is our opinion that, in order to improve the performance of actor applications and their REs, we first have to understand the behavior of these applications. This is, in fact, the first contribution of our work. We analyze some applications and we show that not only the average actor is short-lived (and that those that are long lived are mostly idle) but also that there is a special kind of actor that we call *hub*. Hubs are actors that are involved in many more communications than the average actor. They are in fact the orchestrators of the application execution. We define the set of the actors that exchange messages with a hub actor as the *hub's affinity group*. We then present a behavioral hub-based heuristic that uses the fact that inter-affinity-group communications are rare and that an actor present on a hub's affinity-group was probably spawned by this same hub.

► **A Hierarchical Approach to Actor Runtime Environments** With a better comprehension of the behavior of these applications, our next contribution is the proposal of a set of optimizations for an actor RE. These optimizations try to match the execution of the applications to the underlying hardware platform therefore improving hardware resource utilization. We introduce the concept of *actor home nodes* (essentially this is the NUMA node where an actor heap lies) and explain how it can improve the performance of actor applications on hierarchical memory platforms. Then, using the heuristic developed by the previous contribution and the home nodes, we propose a set of improvements to the load-balancer and initial actor placement policies on these REs.

By introducing our ideas into a state-of-the-art VM, the Erlang VM, we were able to assess the efficacy of our proposal. We describe this work along with the performance results obtained in different platforms for some real and synthetic applications created specifically to test different aspects of the execution.

1.2 Scientific Context

This thesis was developed under an agreement for a joint-degree between the University of São Paulo (USP) and the University of Grenoble. It has been divided in three phases, the first and the last developed in São Paulo, at USP's Institute of Mathematics and Statistics (*Instituto de Matemática e Estatística*, IME), and the second in Grenoble, at Grenoble Informatics Laboratory (*Laboratoire d'Informatique de Grenoble*, LIG).

The first phase involved the basic formation of the candidate, and the research and establishment of the research topic. This work has been carried out within IME's Distributed Systems Group (*Grupo de Sistemas Distribuídos*, GSD). During this phase the candidate received funding from projects Baile[2] and CHOReOS[3]. Baile was a research project funded by HP Brazil and carried out at the IME-USP FLOSS Competence Center in collaboration with HP Labs. CHOReOS was a research project part of the FP7 European program, developed at the IME-USP FLOSS Competence Center as well as several European institutions. These projects allowed close collaboration with researchers of HP Labs, and several European universities and companies.

The second phase includes the period in which most of the research contained in this document was developed. It was also during this phase that the publication of the first results was done (see Appendix A). The candidate worked with the Nanosim (Nanosimulations and Embedded Applications for Hybrid Multi-core Architectures) team. Nanosim members had the opportunity to collaborate with people from research institutions from around the world such as the *Universidade Federal do Rio Grande do Sul* (UFRGS, Brazil), *Commissariat à l'énergie atomique et aux énergies alternatives* (CEA, France), *Institut National de Recherche en Informatique et en Automatique* (INRIA, France), and Université de Yaoundé 1 (Cameroon). During this period, the author received funding from project CAPES/COFECUB[4].

The third phase dealt with the conclusion of the thesis and the publication of final as well as some satellite results. It was done with the GSD group and involved the presentation of our findings in international conferences. The feedback received in

[2]Baile: Enabling Scalable Cloud Service Choreographies – `http://ccsl.ime.usp.br/baile/`. Grant HP-037/11.

[3]CHOReOS - Large Scale Choreographies for the Future Internet – `http://choreos.eu/`. FP7-ICT-2009-5, Grant #257178

[4]CAPES/Cofecub Project #660/10

these venues, along with the work we developed, allowed for the consolidation of the results in the form presented in this document. This final phase was funded by CAPES through a institutional scholarship awarded to the author.

1.3 Thesis Outline

The remaining of this document is organized as follows:

- Chapter 2 presents a quick revision of the concepts and tools involved with the efficient utilization of multi-core and hierarchical multi-core platforms. Then it presents the Actor Model, including some details of the existing REs and applications.
- Chapter 3 outlines our work on the behavior of actor applications. First we characterize actor applications and then present a behavioral heuristic based on these findings.
- Building on the results described in the previous chapter, Chapter 4 outlines our proposal for a hierarchical memory-aware actor RE.
- In Chapter 5 we show our evaluation methodology, including the used platforms, the modifications we did to the Erlang VM and our experimental results.
- Chapter 6 discusses several works related not only to the optimization of actor REs to hierarchical memory platforms but also to the optimization of parallel REs in general.
- Chapter 7 brings our conclusion, including an overview of our contributions, and outlines future works.

In this document we divided the bibliographical references into two different categories. In the first category we list traditional publications such as scientific and academic papers. The citations keys used for these works are defined using the authors' initials and the year of publication. The second category was used for online resources such as product home-pages, software manuals, source-code repositories and press-releases. In this case, the citation keys are simply sequentially numbered by their order of appearance in the text.

CHAPTER 2

Background and Motivation

PARALLEL programming languages have recently been under the spotlight. They are not something new [AV90, IKBW+79]. However, until about twenty years ago they were, in practice, restricted to the High-Performance Computing (HPC) and research domains. During the last years, mainly due to the introduction of multi-core processors, we observe a rekindled interest in these languages, in particular those based on the actor model [KSA09]. These languages allow their user to tap into the processing power of the new multi-core processors with almost none of the hassle that is normally associated with parallel and concurrent programming. Actor based languages are being used in many different areas, but mostly to build highly-available and scalable servers. Some heavy users are Amazon, Facebook, Yahoo and Twitter [CT09, 9].

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. Furthermore, some architectures have an asymmetrical memory hierarchy, thus communication costs[1] between PUs, even those in the same processor, are not constant. NUMA architectures further increase the memory hierarchy asymmetry by adding local memory to each node. The actor model RE, as an additional layer over the operating system, has supplementary information about the application behavior. This information can be used by the RE to make better scheduling and load-balancing decisions.

[1]In this context, an increased cost means higher latency and decreased bandwidth.

We start this chapter with the introduction of hierarchical shared memory multi-core platforms (Section 2.1). In Section 2.2 we outline the basic concepts of actor applications necessary for the development of our work as well as its relation to these challenging hardware platforms. Finally, in Section 2.3, we conclude with our remarks.

2.1 Hierarchical Shared Memory Multi-Core Platforms

Some current highly-parallel processors already feature hundreds or even thousands of cores, while being energy efficient. The execution model of these processors usually follows two different approaches. Light-weight and general purpose multi-core and many-core processors, such as Intel's Atom [10] and Xeon processors, Samsung's Exynos 5 Octa [2], and Kalray MPPA-256 [DML+13], offer autonomous cores and a shared memory execution model that supports POSIX-like threads to accomplish both data and task parallelism. Differently, Graphics Processing Units (GPUs) follow another approach based on a Single Program, Multiple Data (SPMD) model, relying on runtime Application Programming Interfaces (APIs) such as Compute Unified Device Architecture (CUDA) and OpenCL. Here we are interested in the former, *i.e.*, multi-core platforms that support shared memory.

Uniform Memory Access (UMA) platforms, or more generally Symmetric Multi-processing (SMP) platforms, are shared memory platforms that possess at least two identical PUs. Each PU, in fact, can take the form of an entire processor, or the form of a core on a multi-core processor. Intel's Hyperthreading technology [BBH+04] pushes this definition a little bit further providing two execution threads per core, *i.e.*, two PUs per core. A simplified architectural diagram of these platforms is depicted in Figure 2.1.

In these architectures, the single access bus to the main memory can become an important performance bottleneck as the number of cores increases. NUMA architectures appeared as one possible solution to the scalability of multiprocessor SMP platforms. NUMA machines are normally composed of several multi-core processors divided into nodes with their own memory. Each one of the NUMA nodes can be

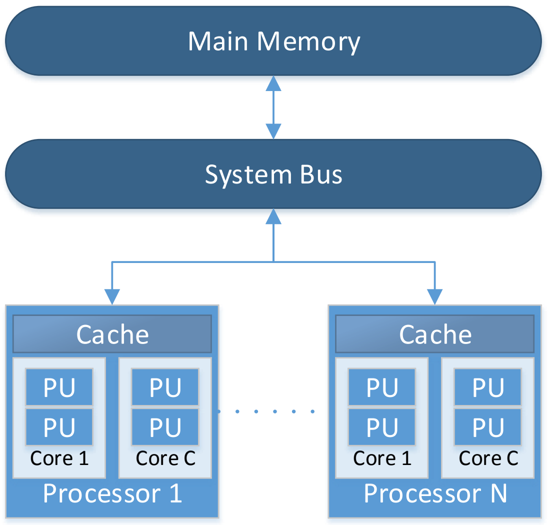


Figure 2.1: A simplified diagram of an SMP machine. In this figure, the machine is composed of N processors with C cores each. Each core has two PUs, totaling $2 \times N \times C$ PUs. Accesses to the main memory are symmetrical across PUs thus the name of the architecture.

roughly seen as an independent SMP machine. However, the difference from an SMP platform is that these NUMA nodes are interconnected by a fast interconnection. This interconnection is used by the system, using hardware support embedded in the processors, to create of a global memory space. This ultimately results in the ability to run regular SMP applications on NUMA platforms with no code adaptations whatsoever. Figure 2.2 brings a simplified diagram of this architecture.

Due to the difference in the performance of the local buses and the NUMA interconnections, there might be a significant disparity between the costs to access distinct parts of the memory (depending if it is local or remote). We call *NUMA factor* the ratio between remote latency and local latency for memory access in these machines. This factor depends not only on the platform itself but also on the chosen NUMA nodes. This happens because the interconnection between the nodes might not be a full graph and, even when this is the case, the latency can change since not every link has the same capacity or utilization levels. Figure 2.3 shows a real NUMA interconnection topology.

Contrary to SMP platforms, the location of the allocated memory is not always local on NUMA platforms. On these architectures, for each memory allocation the OS

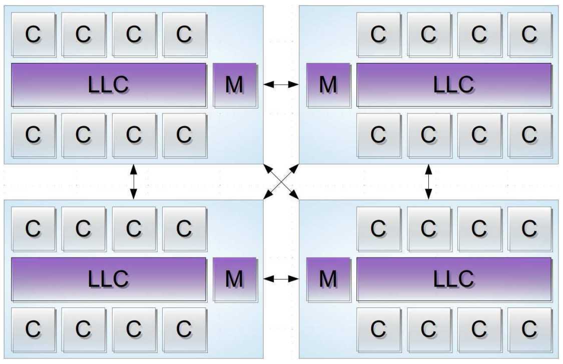


Figure 2.2: A simplified diagram of a NUMA machine with four nodes with eight cores (C) per node. The LLC of each processor is shared by all its cores and the NUMA interconnection is a complete graph. Every core in the same node has symmetrical access to the local memory (M) however they have to employ the NUMA interconnection to access remote node's memories.

must determine on which NUMA node it will be located. The operating system may try to allocate memory in a way that maximizes proximity to the thread/process that will use it. For example, to determine this proximity Linux uses by default a simple policy called *first-touch*. This policy dictates that the memory will be allocated on the NUMA node on which the process/thread that first accessed it was executing. This is a configurable behavior and Linux offers a few other memory allocation policies such as trying to interleave the memory throughout the NUMA nodes.

Accesses to remote regions of memory are not the only operations that suffer from NUMA effects. Storage devices, network interfaces, GPUs, etc, are connected to a specific NUMA node. Therefore, accesses to these devices also incur on performance penalties if the thread performing these operations is not on the same node of the device.

To demonstrate a simple NUMA effect on one of these platforms, we wrote a micro-benchmark on Linux using the `libnuma` library [Kle05]. The benchmark works as follows. First it allocates an array big enough (in our case around 100 MiB) to exceed the storage capacities of the processor caches on the 0-th NUMA node. Then, to avoid hardware pre-fetching mechanisms, we perform a series of non contiguous read-write accesses to its elements and measure how long it takes. We take as a baseline the time needed to run these tests using a thread running on a PU on the

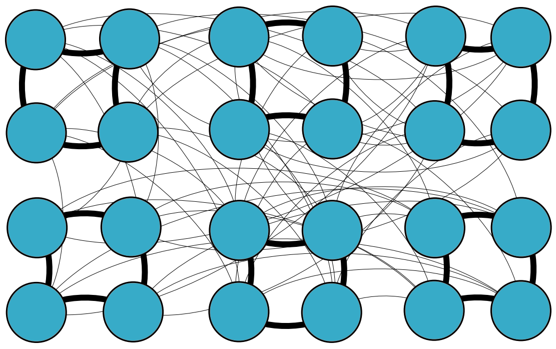
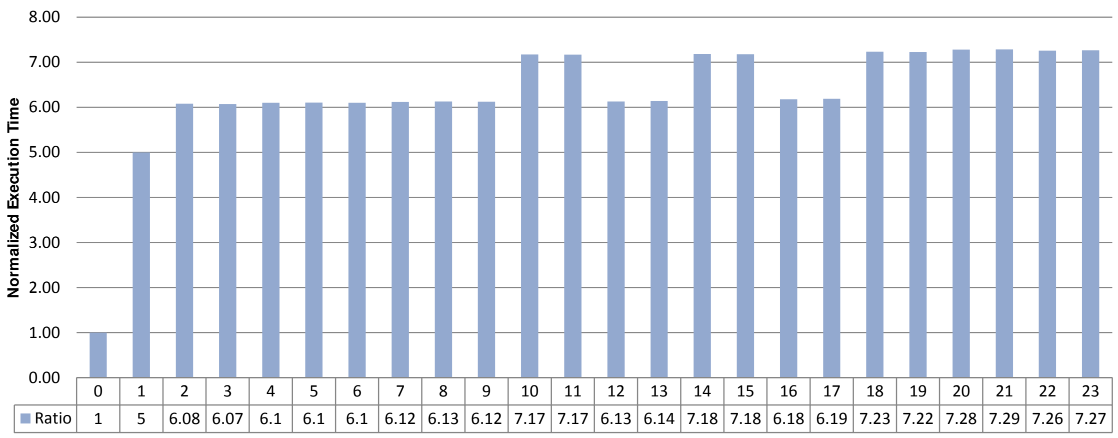


Figure 2.3: Graph representing the real NUMA interconnections of the Altix UV 2000 platform. Each vertex in the graph represents a node and each edge an available NUMA interconnection link. Its topology follows that of a partial-hypercube (since the machine has 24 nodes). Some nodes have privileged connections with other nodes represented in the figure by the thicker edges. A detailed description of this platform can be seen in Appendix C.

same NUMA node where the array is allocated. Then, we repeat the experiment using one PU on each of the remaining NUMA nodes and compare its execution times to the baseline while accessing the same region of memory that was allocated on the first step. Figure 2.4 shows the obtained results using the Altix UV 2000 platform (a full description of this platform can be found in Appendix C). The difference caused by the number of hops between the NUMA nodes can be seen in this figure: the higher the number of hops the longer it takes to execute. In this graph we can also see another characteristic specific to this machine. In this platform, NUMA nodes are also organized in pairs that have an special interconnection between them. Physically these pairs are easily identifiable since they share the same blade on the rack.

Some machines might display NUMA factors higher than 10 [FGM13a]. This big a factor makes the correct placement of processes and memory essential to obtain good performances [CPRA$^+$12a]. Careful placement of memory and processes (also known as process pinning or process affinity) for an efficient utilization of these machines has been a very active field of research [RMC$^+$09, CPRA$^+$12b, MG12, SS12]. APIs to control memory and process placement on SMP and NUMA platforms are available in most OSs such as Linux, FreeBSD, OpenBSD, Solaris and Windows. Among mainstream OSs, Mac OS X is one notable exception. It does not expose an API for explicit process affinity. Instead, it provides an API whose purpose is to provide hints



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
■ Ratio	1	5	6.08	6.07	6.1	6.1	6.1	6.12	6.13	6.12	7.17	7.17	6.13	6.14	7.18	7.18	6.18	6.19	7.23	7.22	7.28	7.29	7.26	7.27

Figure 2.4: Normalized execution time to run a fixed series of read-write operations using an array allocated on the 0-th NUMA node on the Altix UV 2000 platform. On the vertical axis the normalized execution time, on the horizontal axis the NUMA nodes on which the testing thread was executed. There are at least four different access-time classes clearly visible in this figure. Access-times vary according to the NUMA distance between the nodes.

to the scheduler that may, or may not, follow them [11]. In spite of that, to the best of our knowledge, published research on the adaptation of actor REs for these platforms remains scarce.

Fast access to local memory and an increased variable cost to access data on different nodes present challenges not only to the actor model REs but also to any concurrent application. These challenges involve, among others, 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.

2.1.1 Programming Tools

Memory access speeds and bandwidth have been steadily increasing during the last decades. However they have not been able to keep up with the performance improvements of the PUs [McC95]. To avoid the long access times associated to main memory access, chip designers include cache memories into their processors. These memories, although small in comparison to the system main memory, are very fast. So, for data (or instructions) that are often used, the processor can lower the access time, thus increasing its performance. Modern computers have already three levels of cache that, although having different speeds and capacities, are much faster than accesses to the main memory.

On NUMA platforms the maintenance of the caches is more complex than that of an SMP machine. On these architectures not only many PUs do not share the same cache but also they are on distinct NUMA nodes. If two of these PUs are working on the same dataset and one of them performs a write operation it will cause a *cache invalidation* as part of the *cache coherence protocols*. Cache coherence protocols exist to guarantee that each PU always has a consistent view of the memory. During a cache invalidation, the minimum amount of data to be transfered to keep caches synchronized is a cache line. This transfer will be done by either intra or inter-socket interconnections (inside the same NUMA node) or using the NUMA interconnections. This takes time and has a non-negligible impact on the application performance depending on the topological distance between the location of the involved PUs.

To tackle this kind of problems, both automatic as well as manual approaches employ two basic tools: *process pinning* and *memory pinning*. Process pinning consists in defining to the OS, for each process, the set of PUs on which it is allowed to execute. On the other hand, *memory pinning* consists in defining for each allocated region of memory the NUMA node on which it should reside. These operations are normally made available as low-level OS API calls.

Although available in most OSs, these APIs are not portable. But, even if they were, the number of NUMA nodes, the number of PUs, the memory hierarchy and the NUMA interconnections are all platform specific. This means that an application optimization made to a specific architecture might need adaptations if the underlying hardware platform is changed. A project called *Portable Hardware Locality (hwloc)* [BCOM+10] was created to ease part of these problems. This software package provides a portable (across OSs and platforms) API to platform introspection. It can determine the number of NUMA nodes, main memory size, cache sizes, memory hierarchy and sharing, number of cores, etc. It also provides a common cross-platform API to perform process and memory pinning.

By its performance and processing capabilities, NUMA platforms are part of a wide variety of currently deployed mission-critical systems. An important share of these systems is backed by applications written in languages supporting or based on actors. In the next section we describe this model, its realizations, and some of the current uses.

2.2 Actors

The roots of the actor model as a parallel programming paradigm date back to the middle 80's [Agh86]. During the 90's there have been considerable efforts towards the formalization of the model [AMST97, Tal00, MT99]. However, in this document, we are interested in its runtime aspects, *i.e.*, we are interested in the model characteristics that can directly be used to improve the performance of existing systems. We will therefore discuss how current REs realize the actor model with an special attention to the Erlang case. The Erlang RE was chosen because it is one of the most well known implementations of the actor model. Moreover, Erlang has a good market penetration allowing our results to be directly applied to existing applications. For the remaining of this document, when we refer to Erlang VM, or simply the Erlang RE, we will actually be referring to the Erlang BEAM [12].

In this section we describe the basic concepts of the actor model, outlining the high-level aspects of the execution of an actor application. Then, we present some low-level details of current REs, essentially showing how they are built. We are aware that the definitions and explanations given in this section might not exactly reflect those of the original model proposed by Hewitt [HBS73] and latter by Agha [Agh86]. The reason for that is twofold. First, we are interested in providing a didactic and pragmatic approach to actor programming. Second, since our goal is to analyze and to improve real actor REs, our explanations are deliberately much closer to what real actor REs are than to the theoretical actor model.

2.2.1 Actor Programming

Actor application programming is based on very simple principles and, at first, it might be difficult to understand how real applications can be built using this model. First, an actor is able to perform only a few primitive operations. It can only read from and write to its own data, it can send messages, it can selectively receive messages (occasionally waiting until they are received), and it can spawn new actors. Apart from accessing runtime libraries, that is essentially what an actor can do[2]. Although the basic operations available to an actor might be few and simple, it is the way in which these operations are tied together that makes the expressive capabilities of the actor model interesting.

An actor encapsulates its data and behavior. Since access to an actor's data set can only be done by the actor itself, chances of race conditions on data accesses are non-existent. It also means that any communication between actors is done exclusively through message exchanges. To provide actors with messaging capabilities, the actor model establishes the existence of individual mailboxes, one for each actor. Whenever a message is sent to an actor, it is placed in its mailbox. Any actor can send messages to any other actor in the system provided it possesses the receiver address. Actor addresses are opaque structures that are independent of their location. Not having to specify the physical location of a message receiver allows the RE to freely distribute the actors throughout the hardware platform in a way that is transparent to the application.

The delivery of messages is asynchronous. There is no delivery receipt and, for mainstream real systems such as Erlang, there is only a best-effort policy for message delivery[3]. There is no guarantee as to the order in which the messages will be delivered and the receiving actor can pick which messages from its mailbox it wants to process (selective receive). Figure 2.5 illustrates this process.

To keep this message exchange scheme working, each actor has a unique and private execution flow associated with it. This execution flow is represented by the *event loop*. An actor is *alive* as long as its event loop is active and vice-versa. The event loop has two possible active states *executing* or *waiting*. We will deliberately loosen this definition so when we say that an actor is executing or waiting we actually are talking about its event loop. Similarly, we will say an actor is idle when its event loop is on the waiting state.

[2]In Erlang even simple I/O operations (such as writing to a file) are performed using "actor-like" entities called ports. Actors (called processes in Erlang) send messages to a port to write or read data from/to a file or to do any kind of I/O operation.

[3]This is strictly true for distributed message delivery. For the local case most REs, such as Erlang and Scala/Akka, guarantee delivery under normal conditions, *e.g.*, the receiving actor is alive, there is enough memory, ...

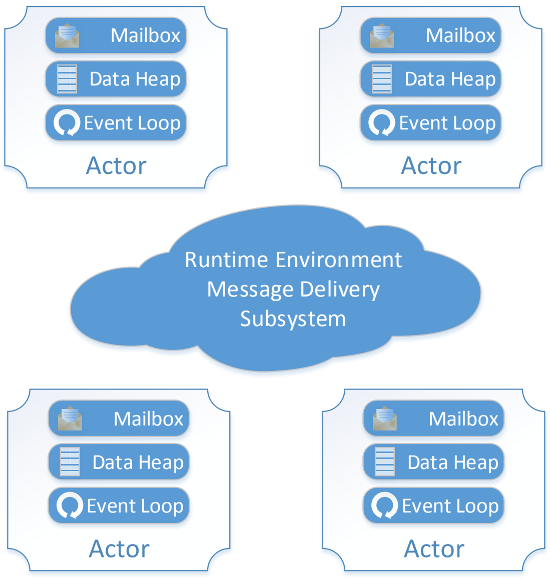


Figure 2.5: Simplified diagram of an actor RE. Each actor is independent of the other actors and is basically composed of a mailbox, a data heap and the event loop. Messages are directly delivered to the mailbox by the RE message delivery subsystem. During the execution of the event loop messages can be selected and moved from the mailbox to the actor's heap to be processed.

Since each actor has its own event loop, messages are processed one at a time. This singular behavior is part of the core of the model. It allows for the non-existence of locks, semaphores or any synchronization construction. When needed, actual synchronization between actors is achieved by creating an actor that controls the shared resource and the remaining actors communicate with it using exchanges of messages. This is similar to how distributed systems, such as MPI, synchronize the different execution lines. Figure 2.6 shows how a simple critical region of code, in this case a shared counter, can be implemented.

An actor is self-contained, the only way we can verify or modify its state is by sending it a message. That explains why, even if an actor also represents an independent flow of execution, typical actor applications have a much higher number of alive actors than the available number of PUs. This also explains why some actors are mostly idle (they exist only to keep states) and why so many actors have a short lifespan (they are created to perform a specific task and then they are discarded).

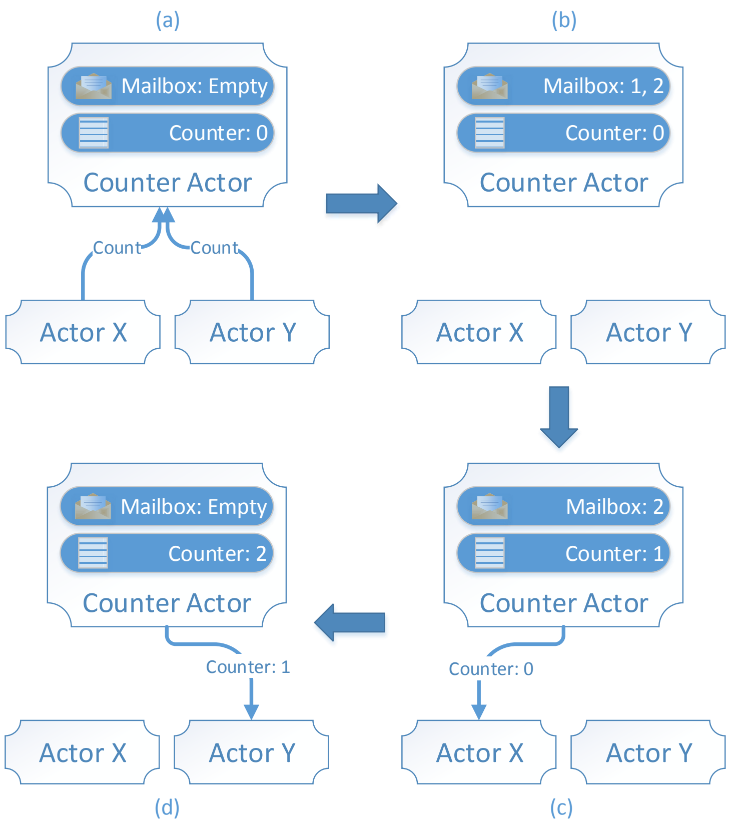


Figure 2.6: A simple exclusive access counter implementation using actors. Beginning at the upper left corner *(a)*, actors *X* and *Y* send a message asking for the counter's current value. Both messages are put in the *Counter Actor's* mailbox *(b)*. The event loop takes one message from the mailbox, processes it and sends a message back to the sender with the current value of the counter *(c)*. The process is repeated until the mailbox is empty and then *Counter Actor* becomes idle *(d)*.

The idleness of actors can also be explained by how actor applications are built. For example, some actor applications, such as web servers and databases, might create an actor to listen to requests over a TCP/IP socket. Whenever a request is received a new actor is spawned to deal with that specific request. Unless the number of requests is extremely high, this listener actor will be mostly inactive. Which brings us to the next common characteristic of actor based systems, the creation of actors is very fast and, contrary to other parallel/concurrent programming tools such as threads, it is usually desirable to create them in vast quantities whenever they are needed.

The means by which current actor REs realize the actor model are discussed in the following sections. First, Section 2.2.2 outlines general execution aspects of actor REs. Then, Section 2.2.3, analyzes the effects these design choices have on NUMA platforms.

2.2.2 Runtime Environments

Erlang [Arm10], Akka [Gup12], and Salsa [VA01] are some examples of the vast number of currently available actor REs. Although each one of these REs has its own peculiarities, most of the runtime aspects we describe in this section can be directly applied to them, be them language or library based.

The abstraction provided by the actor model, although very powerful, willingly takes the application developer away from the underlying hardware platform idiosyncrasies. Thus, the actor RE becomes the responsible for a transparent and efficient use of the underlying architecture. Here we will describe, with a special interest in the Erlang VM, two of the main aspects tackled by these REs: actor scheduling and actor memory management[4].

2.2.2.1 Actor Scheduling

Not only actors represent each distinct flow of execution of an application, but they also keep the state of the system. Therefore it is expected that the number of actors alive in a system to be higher than the available number of PUs. For this reason a time sharing solution for the PUs must be employed. There are two basic distinct approaches REs use to realize the actor model. The *thread-based* and the *event-based* approaches. The main difference between the two is that 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.

[4]In Erlang actors are called *processes*, however, in order to avoid confusion, from now on we will use the term *actor* to refer to each Erlang VM internal process, and the term *process* to refer to each operating system process.

The event-based approach is the choice made by the Erlang VM while Scala [OAC+04] allows the developer to choose between them. Akka, for example, lets the developer go even further by providing the choice (or definition) of distinct approaches on an actor basis [13]. Thus, in some systems, distinct actors might be using, at the same time, thread-based, event-based or hybrid approaches. A nice characteristic of the event-based approach is that 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. On Akka each data structure that represents an actor consumes about 600 bytes while in Erlang each actor uses about 2,500 bytes (including the heap and the stack, although they might be grown if needed). This means that an Erlang application is able to create about 420,000 actors per gigabyte of memory. This is hardly the case for thread-based REs.

Event-based REs mostly work by creating one or more 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 (Figure 2.7). Even though binding schedulers to the available PUs could improve performance by providing, among others, a better use of the processor caches, they are not bound by default in most REs. Although the reasons for this decision are not clear, one possible motivation might be the fact that if the RE is not the only demanding process executing on the machine, process pinning could degrade performance if the threads of the competing applications are bound to the same overloaded PU while the rest of the PUs are free. The rest of this document assumes that schedulers are bound to the available PUs, and that the RE is the only demanding process in the system. Since most high-performance actor REs execute on dedicated machines this requirement is, in most cases, automatically satisfied.

In event-based approaches, 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. When scheduled for execution, an actor 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 actor will be preempted by the scheduler and

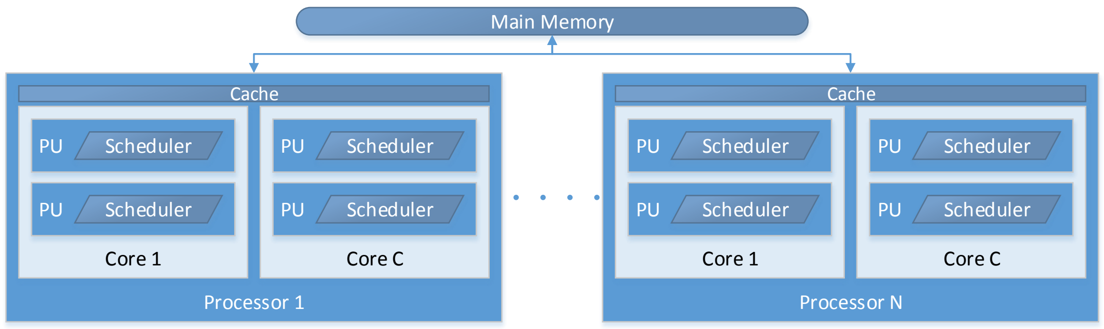


Figure 2.7: Simplified view of the schedulers and their bindings to the machine's PUs. Event-based actor REs work with a pool of threads. These threads, called schedulers, can be occasionally bound to specific PUs to optimize, for example, cache usage.

put back on the run-queue. The scheduler will then schedule the next actor on the run-queue for execution.

Some actor REs allow actors to have different execution priorities. Thread-based approaches simply set different execution priorities for each thread and let the OS handle the processor time-sharing. Event-based approaches on the other hand typically keep one distinct queue per priority. In fact, this gives the user the illusion that there is only one queue from which actors are scheduled according to different priorities. However in reality there are multiple queues (one for each priority) for each scheduler. The Erlang VM is an example of such a RE. It has four distinct priorities (`low`, `normal`, `high` and `max`). Actors with `low` or `normal` priority have their execution interleaved. On the other hand, actors with `low` and `normal` priorities are executed only if there are no `high` priority actors waiting for execution, and actors with `low`, `normal` and `high` priorities will only be executed if there are no `max` priority actors ready for execution.

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, each actor has distinct lifespans. Additionally, actors do not have the same behavior when it comes to actor spawning. In the Erlang VM an actor that spawns many more actors than the application average might cause imbalances in the run-queues since, by default, each actor is placed on the same run-queue of its father. The *initial placement* and *load*

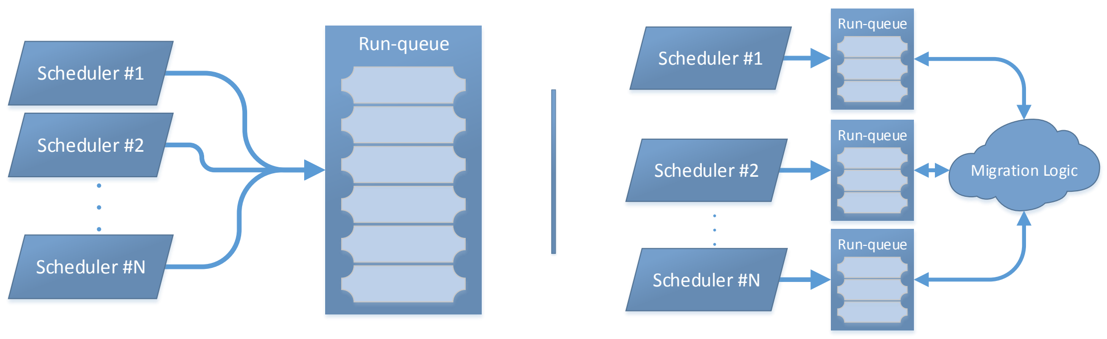


Figure 2.8: On the left a RE with a single run-queue. The schedulers use the same queue to keep all the runnable actors. On the right a RE with multiple run-queues, one run-queue per scheduler. In this case load-balancing and work-stealing strategies are employed to keep the queues balanced.

balancing policies of each actor becomes, therefore, an important runtime aspect of the RE. Figure 2.8 illustrates this process.

To control this imbalance the Erlang VM, similar to methods employed by other REs, employs two distinct strategies: work-stealing and periodical load-balancing. If a scheduler runs out of actors it will steal work from 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 actor receives a fair share of the available PUs. Hence the RE periodically runs load-balancing routines. The load-balancing criteria used to determine which and how many actors will be migrated is implementation dependent, but the goal is invariably the same: make every queue have approximately the same size.

There is yet another kind of imbalance which is typical of underloaded systems. In this case a strategy called *compaction of load* is employed. This strategy will try to use as few schedulers as possible and try to minimize how often schedulers run out of work. The rationale behind it is that a small number of actors in the system makes them more susceptible to bounce between schedulers since any small variation in the number of actors might prompt a migration, for example, by the load-balancer. The compaction of load works by detecting how often schedulers run out of work and, if this frequency is higher than a pre-defined threshold, the RE migrates the runnable actors to a smaller set of active schedulers. The remaining schedulers are suspended. If at a some point the active schedulers are not able to keep up with the work, some of the suspended schedulers are waken up and the load is rebalanced. The reason the RE needs this kind of behavior is related to how actors are treated. Actors are supposed to be platform agnostic, and therefore it makes no sense to bind the execution of an actor (or a group of actors) to a scheduler. On the other hand, the RE has information that can be used to avoid this kind of errant behavior by, for example, trying to keep actors with a high affinity, *i.e.*, that communicate a lot, close together avoiding bouncing between the schedulers.

As any thread-based programs, if two actors are competing for resources there might be an important level of contention. This might have a non-negligible impact in performance. Contrary to thread-based approaches where, in some cases, contention might be solved using process pinning, there is not a widespread solution equivalent to thread-pinning for actors[5]. We call *actor-pinning* the operation of restricting an actor execution to a set of specific schedulers. It is important to note that, this restriction will only be effective if schedulers themselves are also bound to the PUs.

When we compare the OS load-balancing strategies to that of an actor RE, we can promptly find a handful of differences. The first point has to do with fairness and overall efficiency. Most OSs by default will strive to make every available PU busy for most of the time. The rationale here is that by keeping every PU busy then, on average, the system will perform better as a whole. A second point we must take into consideration is the difference in the granularity of the scheduled entities dealt with by an OS and an actor RE. The frequency in which actors are created is typically much higher than that of processes or threads. Not only the frequency is different, but also the number of simultaneous alive actors tends to exceed the number of available PUs by one or two – and sometimes three – orders of magnitude. This forces an efficient actor scheduler to possess a few important characteristics. In such a setting, context-switching must be very efficient, occasionally more efficient than that performed by an OS. Moreover, in a system with just a few PUs and thousands of actors, scheduler time-slices might need to be shorter than their OS counterparts in order to avoid long waiting times. One of the main concerns to be taken into consideration during the development of such a scheduler is that actors are created to do very small tasks while processes are entities that tend to live much longer.

[5]Despite being undocumented and unsupported, Erlang has an actor-pinning API [FGM12] that allows actors to be pinned to a specific scheduler.

Just like an OS process, in order to work, actors also need access to memory. Both actor scheduling and memory management are responsibilities of the actor RE. In this section we described how current actor REs perform scheduling tasks. In the next section we will outline the means by which these RE control the memory used by the actors.

2.2.2.2 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 [FGM13a]. 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 Akka, do not have a dedicated VM so a private heap is harder to implement. Actually, in Erlang, since actors are short lived, many are directly discarded never experiencing a garbage collection during their lifetimes.

Central to the actor model, efficient communication by the exchange of messages is an essential aspect for application performance. Actor REs have two options for the implementation of the underlying message delivery subsystem: message passing by reference or message passing by value. Normally, exchanges by reference are implemented using a shared heap architecture whereas exchanges by value normally use a private heap. In the former, the actual exchanged message is just a container for a memory pointer. In this case the size of the exchanged messages is almost constant independently of the data it refers to. In the latter, the data is completely copied from the heap of the sending actor to the heap of the receiving actor, hence message sizes and delivery costs become proportional to the size of the data being sent.

While a shared heap architecture usually brings the advantage of having constant message passing costs and lower memory usage, it also imposes a larger burden on the garbage collector and therefore higher garbage collection times. Private heaps, on the other hand, offer efficient and simple garbage collection methods at the expense of higher costs for message exchanges and additional memory usage.

In an attempt to conciliate these two models and keep the advantages of both, hybrid models in which only parts of the message are copied (or only some kinds of message are copied) have been proposed and evaluated on SMP platforms [CSW03, JSW02]. In this hybrid model each actor has its own heap in addition to a shared heap used for message exchanges. To avoid an increased complexity in garbage collection algorithms, static code analysis, as well as some runtime analysis, are performed on the sender actor to decide which pieces of data will be placed on the private and on the shared heaps. For instance, the current Erlang VM uses a simplified version of this architecture. It employs a private heap for general message exchanges and a special shared heap for binary messages bigger than 64 B. Additionally, Erlang 5.2/OTP R9B also provided an experimental feature in which actors shared a common heap. However, this feature never received official support and is no longer available [14].

In every situation, however, actors continue to possess private stacks that contain the necessary pointers to access the memory addresses of the relevant data, be them stored on a private or on a shared heap. Figure 2.9 illustrates these three memory organization approaches. From left to right private, shared and hybrid simplified representations of the states before (top) and after (bottom) a message exchange. In every case *Actor A* sends a message to *Actor B*. The message contents are represented by a star. On a private heap architecture, the data are copied from *Actor A*'s heap to *Actor B*'s heap and a pointer to the copied data (represented by a circle) is written to *Actor's B* mailbox. On a shared heap architecture there is a common heap shared by all actors. To send a message to *Actor B Actor A* just writes a pointer to *Actor B*'s mailbox. Finally, on the hybrid architecture, the message contents are on the shared heap. *Actor A* just needs to write a pointer to it on *Actor B*'s heap. In this scenario *Actor A* also has some private data that does not take part on the message exchange. These data, represented by a triangle, remain untouched allocated on *Actor A*'s private heap.

The choices made by the designers of the currently available REs suggest that they were not written with the NUMA architecture in mind. In the next section we present various aspects related to the execution of these systems on these hierarchical platforms.

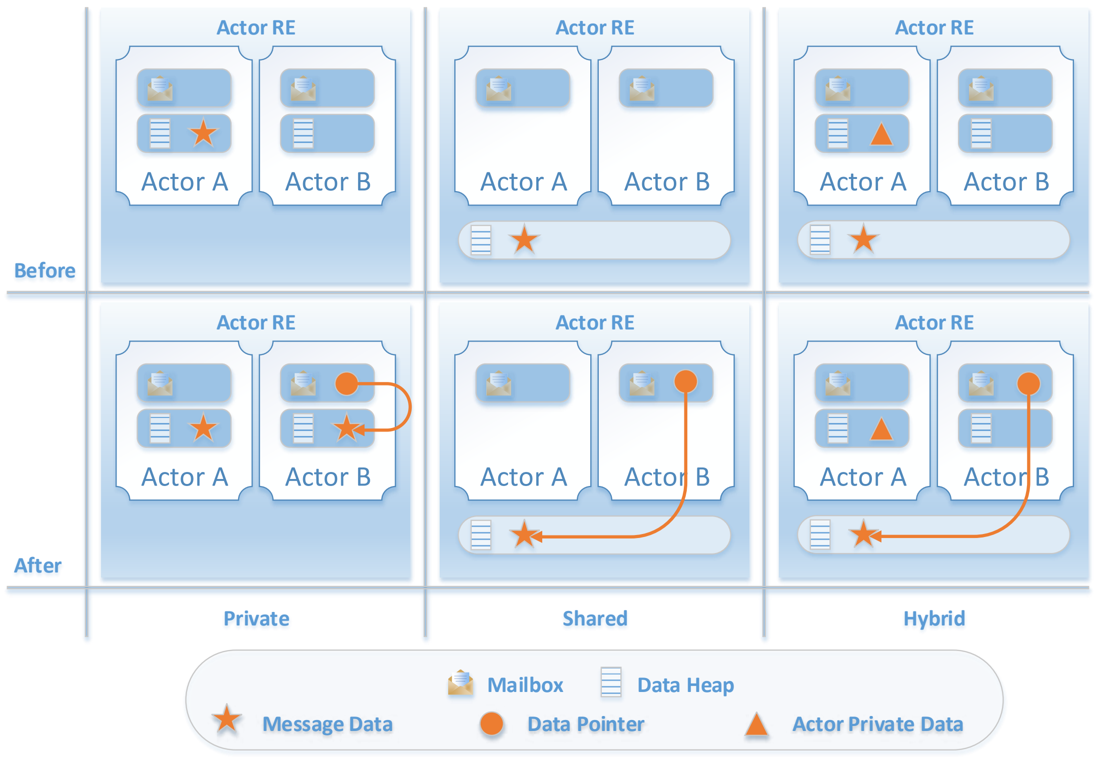


Figure 2.9: Actor heap architectures.

2.2.3 Scheduling and Memory Management on NUMA Platforms

In this section we analyze the design choices available for the creation of actor REs. Once again, we divide our analysis in two main aspects related to an actor execution: actor scheduling and memory management. For each one of these aspects we first describe the base ideas. Then, with a NUMA mindset, we outline their strengths and weaknesses to finally state the reasons why some aspects are more adequate than others for execution on these hierarchical platforms.

2.2.3.1 Actor Scheduling

One of the most important aspects an actor RE has to deal with for the execution of applications is actor scheduling. RE developers have basically two design choices for the actor scheduler: thread-based and event-based. In the thread-based approach each actor is represented by an OS thread. In this case the RE sees itself free from any task regarding scheduling. It directly uses the features provided by the underlying OS. A natural solution considering that each actor is an OS thread. Event-based approaches, on the other hand, have to deal with these issues themselves. They often employ a thread-pool that executes actors from a run queue. In this case there is not a direct relation between the actors and the OS threads. We now describe some of the differences between these two approaches in more detail.

► **Resource Consumption** Besides suffering from a higher context-switching overhead, thread-based actors also tend to consume more memory than their event-based counterparts. Even if threads share the same memory with their processes, their stack is private. By default, on Linux/x86-32 the default stack size is 2 MiB [15] (however this value may vary from 98 KiB to 32 MiB depending on the architecture). Current OS and hardware platforms are able to efficiently create and manage up to a few thousands threads per process, while typical event-based actor REs are able to deal with hundreds of thousands of actors. Not only the amount of memory needed for maintaining a thread-based actor is higher but also the time needed for its creation is more important [HO09].

► **Execution Control** General OS schedulers aim at fairness and overall hardware utilization maximization. Even if thread placement and scheduler parameterization OS APIs exist, the choices made by the OS scheduler can only be influenced up to a point. Proposals for the transparent and efficient management of threads and memory on NUMA architectures by the OS exist [SS12]. Unfortunately not only are these tools still under active development but also they fail to profit from the distinct behavior of actor applications. The main characteristic of thread-based REs is their simplicity of implementation and execution. They delegate all the responsibilities to the OS and the RE deals only with what is strictly necessary to make the actor model work, *e.g.*, the message delivery subsystem. However, to use this kind of OS API the RE needs to monitor the execution of the actors and issue all the necessary API calls to parameterize the OS accordingly. In this scenario all the simplicity, mark of a thread-based REs, is essentially forfeited. On the other hand, for this specific case, an event-based approach could actually be simpler. All the information needed by the RE scheduler is already available and controlled by the RE itself so there is no need to create an additional level of actor monitoring and control.

► **OS Tools Support** Application developers employing regular OS processes and threads have at their disposition a variety of debugging, profiling and tracing tools such as Valgrind [NS07] and OProfile [16]. Actor application developers relying on thread-based REs can directly take advantage of these tools since there is a direct relation between actors and threads. In event-based REs the use of this kind of tool is much more difficult since the link between an actor and a thread is constantly changing. Moreover, optimizations at the OS level [SS12] are no longer as efficient for the behavior of the thread is constantly changing (because of the different behavior of each actor).

In Chapter 3 we show that actor applications tend to create an important number of actors with short lifespans and that their creation might happen in bursts. Typical thread-based actors take much longer to be created than event-based actors. This renders the use of applications that naïvely employ algorithms similar to MapReduce inefficient due to increased ratio between the actor creation overhead and task processing time [HO09]. The concurrent number of actors, their short lifespans and the frequency in which they are created are some of the reasons that indicate that an event-based approach might be a better fit for scalable actor REs on hierarchical platforms.

This conclusion has, however, to be carefully considered. The application characterization we present in the next chapter, was based on Erlang applications that use the event-based approach. Thus it is only natural that the results we found correspond to the underlying VM architecture. On the other hand, when we consider the actor model as it has been defined, we see that the model, in fact, encourages this kind of practice. Actors have a finer granularity than threads or processes. Some argue [Arm07, HO07] that a scalable application is the one that is able to create an arbitrarily large number actors and, in a broader context involving a large set of programming models, some believe that to be truly scalable a programming model must present characteristics such as overdecomposition (many times more “schedulable objects” than the number of PUs), migratability and asynchronicity [Kal13].

Nevertheless, these are not the only reasons encouraging our choice for an event-based approach. In addition to the points we just mentioned, the use of an event-based approach allows us to apply several RE level optimizations (*cf.* Chapter 4) that would be very difficult to employ otherwise. Unfortunately, the downside of this choice is that we end up loosing support of existing OS level tools.

Single-queues *vs.* Multiple-queues 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 [Lun08]. 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, a single-queued solution does not preserve soft-affinity.

A scheduler is said to preserve *soft-affinity* if it does not migrate processes unless it becomes necessary. Both Linux's and Erlang VM's schedulers have this characteristic. On the other hand, multiple-queued approaches usually 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 solution is the most appropriate for an actor RE on a NUMA platform. Table 2.1 summarizes the described differences between each distinct approach.

Similarly to what we have done in this section, the next section analyzes which memory management characteristics are the most suitable to ensure good execution performance on NUMA platforms.

	Thread-based	Event-based	
		Single Queue	**Multiple Queues**
Resource Consumption	High	Fair	Fair
Execution Control	Good	Good	Excellent
OS Tools Support	Good	Limited	Limited
Scalability	Limited	Good	Excellent
Soft-Affinity	OS Policy	No	Yes

Table 2.1: Comparison between actor RE approaches

2.2.3.2 Actor Memory Architecture

Heap allocation is an intrinsic task related to the spawning of a new actor. In most REs, the heap of an actor is allocated by the scheduler of the parent actor. This means that the scheduler responsible for the execution of the parent is also responsible for the allocation and copying of the spawned actor's 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 OS can employ several different policies to memory placement. Linux, for example, uses by default a first-touch policy. For the Erlang RE this means the spawned actor heap location will be the node where the scheduler that created it was running. We define this location as the *actor's home node*.

It is important to note that home nodes are not definitive. Take for example an actor that, for whatever reason, was migrated to a scheduler lying on a NUMA node different of its home node. 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 actor was migrated is not on the same node as the actor'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 depends on the current actor's location and home node.

Figure 2.10 depicts this behavior. In this simplified example three actors are executing on a NUMA machine composed of two nodes. *Actor A* is being executed

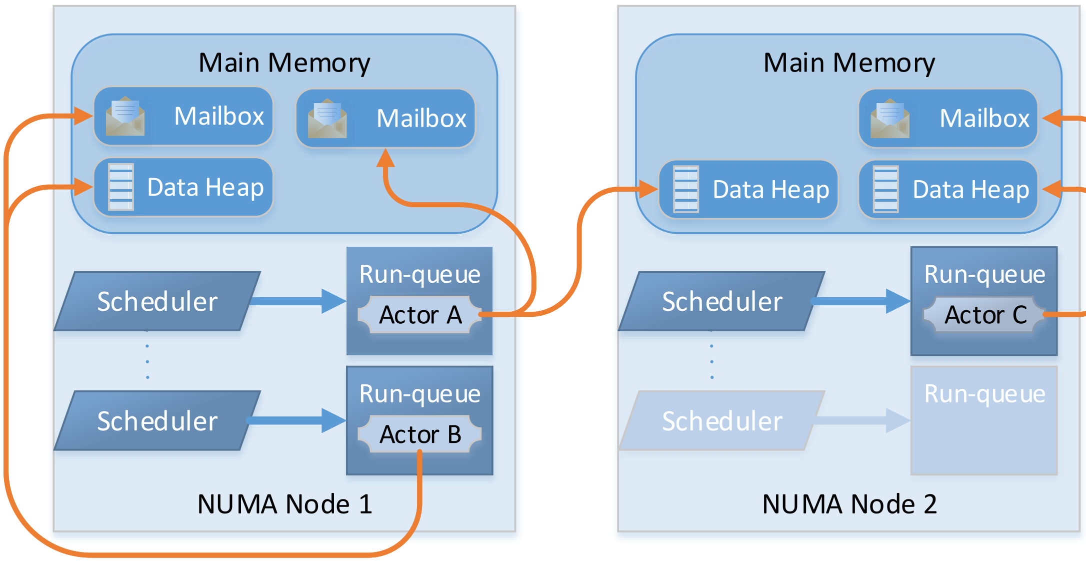


Figure 2.10: Schematic diagram of actors with the location of their heaps and mailboxes.

by a scheduler running on the first NUMA node. Its mailbox is located on the same NUMA node however its heap is on the second NUMA node, *i.e.*, its home-node. A migration followed by an increase in the size of the mailbox might explain why its data structures are spread throughout the machine. Another possibility would be its creation on the first NUMA node followed by a migration to the second NUMA node, an increase in size of its heap and then a migration back to the first NUMA node. For actors *B* and *C* the location of their heaps (home nodes) and mailboxes coincides with the location of their current scheduler.

On SMP platforms shared and hybrid heap architectures can have a small degrading impact on cache locality, however, when the receiving actor accesses the message contents, it is still a local memory access with low overhead. On the other hand, on NUMA architectures a message sent by an actor located on a different NUMA node of the receiving actor will cause a remote memory access by the receiving actor. On a worst case scenario, depending on machine load and access patterns, at each time the receiving actor accesses the data a new remote access might occur. In this case, the receiving actor will always have to pay the costs of the memory transference between the message's origin NUMA node and the destination's NUMA node. Private heaps guarantee that this cost is payed only once. Shared and hybrid heaps on the other hand do not offer this guarantee.

While adaptations to the hybrid scheme to avoid some of these communication costs are possible, they might also become very complex. This complexity arises as a result of actor migrations across different NUMA nodes of the machine. If the migrated actor has pointers to the shared heap, these pointers would have to be followed and the data copied into the actors private heap to avoid execution performance degradations on the new actor's location. The analysis of the pointers and copy of the data would have to be payed anyway in this scenario and, therefore, we believe that the private heap approach, *i.e.*, message passing by value, is the most appropriate choice for an actor RE on a NUMA platform. It is simpler and offers better performance guarantees even if it is not so memory efficient and in the local case it has a higher communication cost. That cost, in fact, is not an additional cost we would be adding to the communications since it is exactly the same cost already being payed, for example, by current Erlang VM users.

As a direct consequence of our conclusion, REs performing actor migrations between NUMA node boundaries need to decide what to do with the data heap of the actors, *i.e.*, keep it untouched (as it is currently done by Erlang) or migrate it to the new NUMA node. Data heap migrations can be expensive depending on their sizes and on the frequency in which actors are migrated. We will revisit this problem in Chapter 4 in which we discuss load-balancing mechanisms.

Actor applications are present in several mission-critical systems running on NUMA platforms. In the next section, we present some notable contexts in which actor applications are inserted and how they are being employed.

2.2.4 Applications

Actor applications are being used in a wide variety of situations such as databases, web servers, simulators and automation tools. Nearly all use cases involve several simultaneous lines of execution, thus a good match for the characteristics of the model. The list of applications we provide below does not intend to be a comprehensive one. Our intention is to demonstrate some interesting use cases and the context in which these applications are inserted.

Twitter [17] employs Scala in many of its internal projects, be it as a full service or as a library. Their current use goes from XML processing libraries to social graph data storage, people search services and tweet streaming over HTTP services.

CouchDB [ALS10][18] and Riak [19] are examples of databases written in Erlang. CouchDB is a NoSQL document-based database. It uses MapReduce [DG08] (controlled by Erlang and executed in JavaScript) to process queries. Riak is a distributed key/value NoSQL database. It also uses MapReduce to process queries however it allows queries to be written both in JavaScript as well as in Erlang. Riak's market share is quite high, with almost 25 % of Fortune 50 companies as its users [20] including Yahoo!, Dell, Github and Best Buy [21]. Some other notable examples of databases written in Erlang are Amazon SimpleDB [22] and Scalaris [SSR08]. Amazon SimpleDB is a commercial, pay-per-use, closed source, cloud-based NoSQL database. Scalaris is a transactional distributed key-value store backed by a peer-to-peer architecture.

Yaws [Vin11] is a web server written in Erlang. It takes advantage of the event-based approached used by the Erlang VM to create a new actor for each received request. In an informal comparison of performances [23] it has been shown that while Apache Web Server does not scale past 4000 simultaneous connections, Yaws was capable to keep responses at an acceptable level past 80 000. Scalable and automatic fail-over Erlang web servers are also used at Facebook to control chat interactions between its users [7]. Similarly, WhatsApp chat servers are also written in Erlang running on FreeBSD [5].

Recently, Chef, a cloud-focused configuration automation framework, had its server, that was originally written in Ruby, rewritten in Erlang [24]. The alleged problems with the old implementation were the lack of scalability and an excessive use of memory. Facebook [25] uses Chef to control part of their data centers.

However, not all applications written using actors are servers. Sim-Diasca [26] is a Discrete Event Simulation (DES) engine written in Erlang. It focuses on the parallel distributed execution of huge simulation cases, while maintaining causality and total reproducibility. *Électricité de France S.A* (EDF) (a French electric utility company, and the world's biggest energy producer), developed and used Sim-Diasca to simulate the performance of the communication infrastructure to be used by energy distribution smart-grids at very large scales [SKZ+11]. ErlangTW [TDM12], in the same application category, is another DES engine entirely written in Erlang that uses Time Warp [Jef85] techniques for parallelization and thus take advantage of the actor model to profit from the newest developments of multi-core architectures.

There are also some embedded systems that use actors. Some of Ericsson's high performance network equipment, such as the AXD301, run on Erlang. The AXD301 is a 160Gbps ATM switch that has more than 1.5 M lines of code written in Erlang (plus 500 K lines of C/C++ used to control low level/device drivers) [CT09].

2.3 Concluding Remarks

Lately, the actor model has seen a considerable growth in its popularity. The reason for that might be the increased availability of multi-core processors and the difficulty commonly associated with the traditional concurrent and parallel programming tools such as POSIX Threads. Based on simple asynchronous message passing, the actor model presents itself as a possible substitute for these traditional tools, without the hassles commonly associated with them. It also brings some advantages such as transparent distribution, modeling [SJ11] and verification [TKL$^+$12] of applications.

Current REs are highly optimized for SMP architectures. These optimizations have reached a point in which seldom does an application developer need to care about the underlying hardware details. However, as the actor model gains in developer adoption, so does its uses on larger architectures such as NUMA. Applications running without an optimized RE for this hardware are not, therefore, performing as well as they could.

In this chapter we briefly described current mainstream hierarchical multi-core architectures, the actor model and how it is currently employed. We have shown some details that can be taken into consideration to improve the performance of these REs in this context. In the next chapter, we will explore these details with much more attention in order to propose, in Chapter 4, a set of improvements inspired by these observations.

CHAPTER 3

Understanding Actor Applications

CHAPTER 2 discussed some characteristics of the actor model. We described, among others, features such as communication by asynchronous message passing, non-existence of shared memory between actors, and transparent parallelization/execution of actors by the RE. In this chapter we will look further into the behavior of the applications themselves, *i.e.*, we are no longer only interested in the REs but we are also interested in how these applications are built and how they work. In order to do that, this chapter starts by analyzing the most basic aspects of an actor application execution.

First, we will describe aspects such as actor lifespan, message sizes, communication costs and communication graphs. In other words, we will draw a simple – yet informative – picture of the general execution parameters that an actor RE has to deal with. Next, we will introduce some higher level behavioral concepts such as *hubs* and *affinity groups*. For this analysis we assume that the applications employ exclusively the actor model, *i.e.*, actors in these applications communicate exclusively through the RE messaging services and that no other concurrency mechanism (*e.g.*, POSIX threads) is used. Erlang applications automatically satisfy this constraint since the Erlang VM only offers the actor model as a means to achieve parallel processing. However, for some other actor applications such as those written in Scala this is not always true. There are several reasons why a developer might choose not to use exclusively the actor model throughout the application. Among those we can highlight limitations of the actor libraries themselves or ease of implementation [TDJ13].

The remaining of this chapter is organized as follows. Section 3.1 describes the applications considered during our analysis. General execution aspects are dealt by Section 3.2 and Section 3.3 describes the varying costs of communication between actors on NUMA platforms. Next, Section 3.4 presents a higher level analysis of application execution introducing the concept of hubs and communication affinity groups. Finally, we conclude in Section 3.5.

3.1 Analyzed Applications

Some aspects of the actor model can be directly used to improve the performance of current 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 easily extractable from the RE that can be used to improve its own scheduler and load balancer decisions. This section presents several of these aspects. Most of them are presented alongside concrete examples and measurements. The data used for the examples were obtained from traces of executions of real Erlang applications and benchmarks using a slightly modified Erlang VM based on version R15B02. These modifications were done so that we could efficiently trace the application's executions. The RE with these new features as well as the hardware platform used for the experiments (NUMA 32) are further described in Chapter 5.

Erlang was chosen because it has an open source VM and is frequently used as a comparison baseline in terms of performance and implementation for many different actor REs [KSA09, HO07, SM08, SLVW08]. Moreover, since Erlang's realization of the actor model is close to the formal definition of the model, it is our belief that the observations and eventual enhancements we make to this RE are also applicable to other actor REs.

To illustrate the concepts described in this section we traced three real applications and three benchmarks taken from the BenchErl [APR+12] benchmark suite. The reasons that motivated the choice of these three applications are three-fold. First, they should be open-source, so that we could analyze and modify them as necessary. Second, since we want to measure the communication performance between actors, with the exception of CouchDB, we looked for applications in which I/O was not an important performance factor. Although CouchDB presents itself as an application with many I/O operations, it was chosen because it employs MapReduce algorithms providing our analysis with interesting communication graphs. Third, applications' actors should communicate primarily (if not exclusively) using the RE provided message exchange platform. We now briefly describe the chosen applications[1].

- **CouchDB** We used Yahoo! Cloud Serving Benchmark (YCSB) running against CouchDB as real application study case. CouchDB [ALS10], is a NoSQL document-based database written in Erlang. YCSB [CST+10], is a database benchmark originally created to measure the performance and scalability of cloud serving systems. YCSB already supplies some common workloads as well as it allows for the creation of new customized workloads. It also provides information about the latency and the throughput of a database system. CouchDB stores documents in JSON format and uses a MapReduce strategy for queries through a RESTful HTTP interface. Results shown in this section were obtained using CouchDB version 1.3.0 and YCSB version 0.1.4.

- **Sim-Diasca** Already concisely described in Chapter 2, this application is a DES engine that can perform simulations on local and distributed environments [26]. The communication graphs we present in this chapter were obtained through the execution of the simulator using the City Waste Management Simulation [Bou13] with the default parameters.

- **ErlangTW** Similar to Sim-Diasca, this application is also a DES simulator engine [TDM12]. It uses the TimeWarp synchronization approach instead of the timed-steps approach used by Sim-Diasca. To produce the communication graphs we show in this chapter we executed the simulator using the PHOLD Benchmark [Jon86].

- **BenchErl** This benchmark suite aims at assessing how well the Erlang VM scales when additional resources, such as PUs, are added to the system [APR+12]. It comes with a set of benchmarks that test different aspects of an application execution such as messaging, processes spawning, and scheduling. In the

[1]A more detailed description of Sim-Diasca, ErlangTW and BenchErl are given in Chapter 5.

scope of this section we have chosen only three benchmarks. The choice was motivated by the illustrative properties of these benchmarks in relation to the concepts being introduced. The chosen benchmarks are: `ehb`, `orbit_int`, and `big`. `ehb` is the Erlang's version of the Hackbench [27] Linux benchmark for schedulers, `orbit_int` is a Distributed Hash Table (DHT) with variable costs for key insertion while `big` assesses message delivery performance on a many-to-many message exchange setting. Chapter 5 contains a more detailed description of these benchmarks as well as a description of the BenchErl suite.

For the evaluation of some aspects of applications' executions, we created our own synthetic benchmarks. These micro benchmarks are explained on an individual basis in the text. In the next sections, we investigate the behavior of actor applications to better understand simple execution characteristics as well as higher level aspects of their execution.

3.2 General Execution Characteristics

Every application has specialized actors to perform distinct kinds of work and 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 short and long-lived actors.

CouchDB, for instance, creates many short-lived actors. Actually 99.5 % of the actors live less than 1.5 s and 88.9 % less than 0.1 s. Figure 3.1 depicts the results of the execution of YCSB against CouchDB.

The real proportion between short and long-lived actors is application specific. As a counterpoint to CouchDB, Sim-Diasca creates actors that live for most of the application execution. However, we can still draw some conclusions about actor REs from this simple example.

First, the actor RE must be very efficient for the execution of short-lived actors, otherwise applications such as CouchDB would not perform well. For short-lived actors, the decision of the initial actor placement, *i.e.*, the choice of the scheduler in which the actor is going to be executed, must be fast. If not the RE would impose a considerable overhead to the execution. The current Erlang VM places the

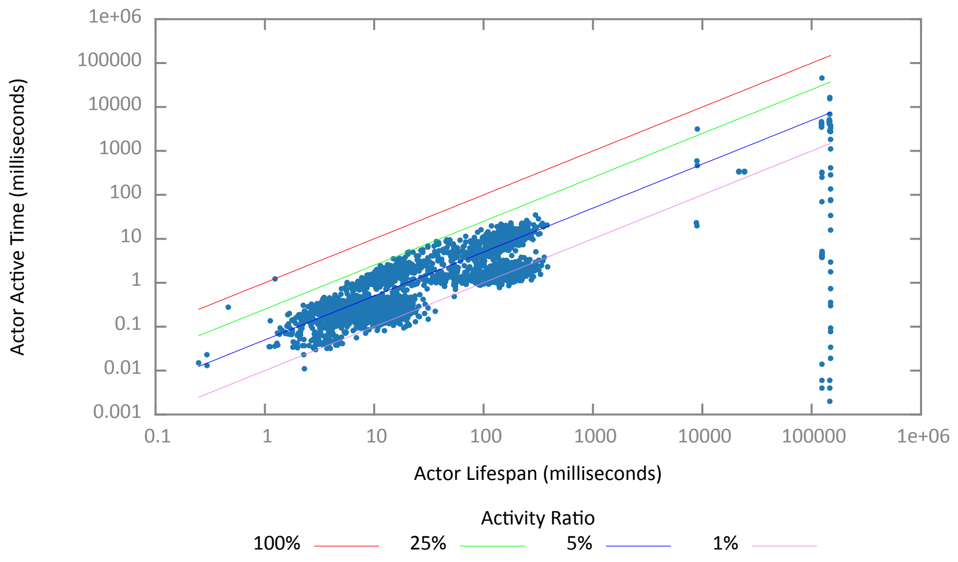


Figure 3.1: Actor lifespan vs. Actor Processing Time. Data for this figure were obtained from execution traces using YCSB against CouchDB. The diagonal lines delimit distinct levels of activity.

newly spawned actor on the same scheduler of its parent (therefore the decision and placement are fast). However this simple strategy might create some imbalances between the queues. These eventual imbalances are only dealt with afterwards, first through work-stealing and then at the next load-balancing round.

Second, typical short-lived actors created to perform specific tasks are (or will try to be) active for most of their lives. However, when we look at their activity ratios, we notice that the vast majority of these actors is inactive for most of their lifetimes. Figure 3.1 depicts this behavior. The diagonal lines delimit different activity ratios for actors created in this CouchDB's test execution. It is immediately noticeable that many actors present an activity ratio below 25 %. Actually, the average activity/lifespan ratio is only 26.74 %. Moreover, some actors have very long life-spans, living for over 140 s. This timespan represents the full execution time of the application. Most of these actors are, however, below the 1 % activity ratio. The reason for this apparent contradiction is two-fold: state and supervision actors, and scheduler time-sharing.

State actors are created to keep the state of the system. In this sense, a state actor can be quite similar to an object in an object oriented programming language where an object encapsulates its state. For this reason, these kind of actors may lie dormant for long periods of time until the data they keep are requested. Supervision actors are responsible for keeping the system running in case of exceptional events such as an unexpected actor death. These actors, created by the application itself often with support of runtime libraries, are present in Erlang [28] and Akka [29] for example. A supervisor is responsible for spawning, killing and, of course, supervising other actors. Supervisors are arranged in a hierarchical structure, which establishes supervision responsibilities between actors. When a supervisor spawns a new actor or becomes responsible for an actor, it gets linked to this actor. Should the supervised die or experience any kind of malfunction, the supervisor will be notified and then can take appropriate action. If a supervisor itself experiences some kind of problem, its supervisor (the one just above it in the supervisors hierarchy) will be notified and can therefore act accordingly. Supervisor actors wait for exceptional events in order to act, therefore they are naturally inactive for most of the time.

Scheduler time-sharing is a necessity for efficient parallel REs, for actors are created in quantities normally vastly superior to the number of PUs on the machine. Therefore some actors, specially short-lived ones, might need to wait for their time share of the processor for a significant part of their lives, specially during actor creation bursts or when the system is loaded.

Third, partly as a natural consequence of the second conclusion, actor REs must be able to deal with copious amounts of actors and with their creation in bursts. The MapReduce [DG08] model, used by CouchDB, Riak [19], and many other actor applications does exactly that: it creates many short-lived actors in a short period of time. Figure 3.2 shows the evolution in time of the number of alive actors on the RE. Even if the underlying hardware platform has only 32 PUs, the number of alive actors is, for most of the execution, above 160 for CouchDB, 180 for Sim-Diasca, and 60 for ErlangTW. It is worth mentioning that these figures can be easily changed depending on the application input. If, for example, Sim-Diasca's simulation were initialized with the *small* dataset instead of the *tiny* dataset, it would have created more than 2000 actors. Similar changes to the input of CouchDB and ErlangTW can be done to produce the same effect.

Once again, the actual number of concurrently alive actors is application and input dependent. Here, our intention is to show that efficient REs will need to be

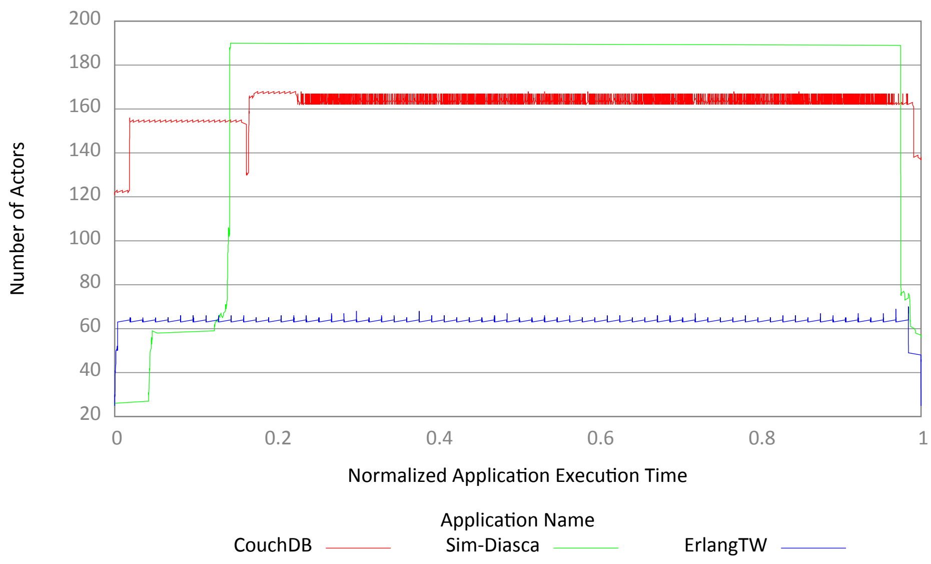


Figure 3.2: Evolution of the number of alive actors in the RE during the execution of CouchDB, Sim-Diasca and ErlangTW.

able to execute on platforms that do not have as many PUs as its number of alive actors. Thus, an important share of these actors will be either waiting for their turn to use the processor or just sleeping, waiting, for example, for messages or for some I/O operation to complete.

The data obtained by these experiments and characterization of actors by levels of activity, lifespans and creation patterns derived from these data, give us the opportunity to improve currently existing actor REs. This can be done using load-balancing algorithms that take these aspects of execution into consideration to take better informed migration decisions.

3.3 Actor Communication Costs on NUMA Platforms

In principle, every communication on the actor model is based on message passing. How it is actually realized 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. Contrary to SMP machines, 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. In these platforms, communication costs can easily become one of the determining factors of the application performance [CPRA+12b]. Figure 3.3 shows, on a NUMA platform, the performance penalty incurred to send messages of different sizes considering the cost between actors on the same PU, *i.e.*, sharing the level one cache, as the baseline. For smaller messages inter-node performance can be more than seven times slower while for bigger messages performance is about half that of the baseline. For the intranode case, performance is about three and two times worse for small and big messages respectively.

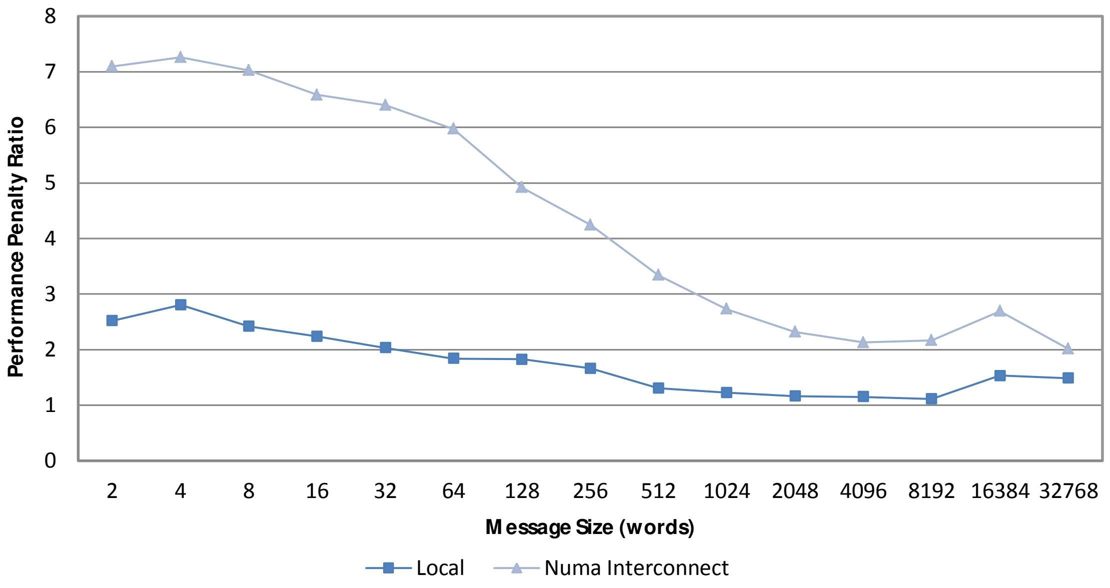


Figure 3.3: Intra and inter-node performance penalty ratio associated to messages exchange of different sizes. The baseline considers the time needed to send messages between actors that lie on the same core. Tests were performed on the NUMA 32 platform. Confidence intervals (95 %) are too small to be visible in this figure and are therefore not plotted.

Our experiments with real applications show that in general messages are small. For the tested applications, more than 90% of the messages were up to 4096 B. Table 3.1 brings the actual average sizes for each application. With the exception of `orbit_int`, the average message sizes did not significantly vary depending on the input. These average message sizes highlight the importance of good actor placements, since small messages sizes are those in which the proportional time difference to send messages between different PUs is the greatest. Table 3.2 summarizes the time needed to send messages for distinct message sizes using three different actor placements on the NUMA 32 platform.

Application	Average Message Size
CouchDB	663 B
Sim-Diasca	1088 B
ErlangTW	960 B
ehb	1024 B
big	2112 B
orbit_int	832 B – 4608 B

Table 3.1: Average message sizes for the evaluated applications and benchmarks.

Words	L1	L3	Slowdown	NUMA	Slowdown
2	1.31	2.47	1.89X	8.57	6.56X
4	1.33	2.55	1.92X	8.80	6.61X
8	1.38	2.62	1.89X	8.99	6.49X
16	1.51	2.67	1.77X	9.32	6.17X
32	1.65	2.89	1.75X	9.99	6.04X
64	1.93	3.32	1.72X	11.10	5.75X
128	2.52	4.73	1.88X	12.64	5.02X
256	3.70	5.60	1.51X	14.87	4.02X
512	5.37	6.95	1.29X	16.89	3.15X
1024	8.97	10.60	1.18X	22.62	2.52X

Table 3.2: Time (μs) needed to send messages of different sizes (words of 64 B) between actors, with placements sharing L1, L3 and with no shared levels of memory on the NUMA 32 platform. Columns "Slowdown" show how many times more it takes to send the message when compared to the best (L1) time possible.

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). This might have a serious impact in the performance. In order to show one of these effects, we created a simple artificial application. This synthetic benchmark is intended to demonstrate the impact of a bad process placement on the processor caches. To this end, we used an instrumented version of the Erlang VM (better described in Chapter 5), and Performance Application Programming Interface (PAPI) with some custom made software that works in tandem with the modified VM. PAPI allows us to read the processor hardware performance counters, including the one that tracks the number of cache misses. The modified VM allows us not only to pin actors to specific schedulers but also to correlate the messages sent to the number of cache-misses. The benchmark itself works as follows. First, we create two actors that exchange messages of varying sizes (from 16 to 4096 B). Then, using the modified VM, we pin the execution of these two actors to two schedulers bound to PUs on the same core (closest possible placement with regards to the caches). Afterwards, while still measuring the number of cache misses, we move one of these actors to a PU on a distinct core on the same processor. Figure 3.4 shows the obtained results. This minimal migration, *i.e.*, just moving the actors to another core on the same processor, caused approximately 1000 times more cache misses than the optimal placement.

In order to do an efficient use of the hardware, the actor RE must be able to take into consideration the hierarchical memory architecture of the machine at hand. The time needed to send a message between actors running on different PUs can vary almost one order of magnitude. Therefore, depending on the application and on the placement of the actors, communication costs can become a very important performance bottleneck. This is one of the reasons why process or thread pinning can be so effective. Usual pinning techniques take advantage of the application developer's knowledge to appropriately map threads to cores, since the developer knows which threads should be placed together. Even if such possibility already existed and was widely used for actors, it would still be a handcrafted solution tied to specific machines and applications.

While analyzing application execution traces we realized that some actors communicate much more than the average actor. There are also actor groups that communicate a lot amongst them but not much with actors outside that group. In the next section we will discuss these actors further.

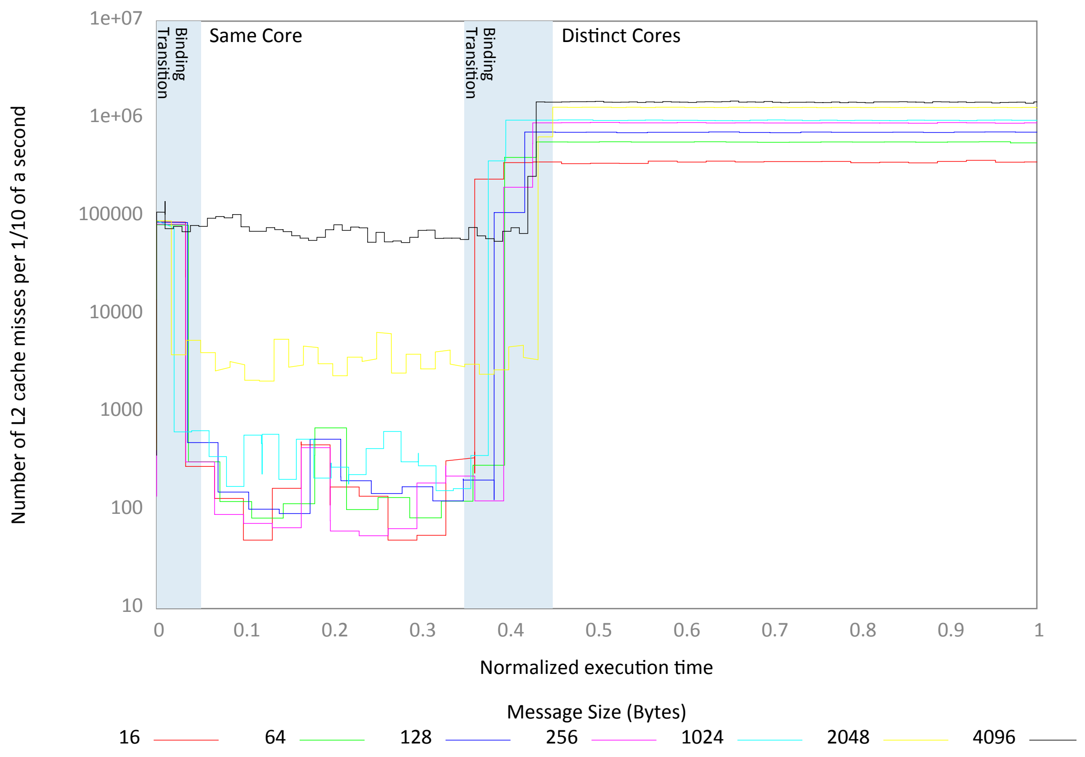


Figure 3.4: Number of L2 cache misses per 1/10 of a second caused by sending messages of different sizes using two distinct sets of PUs. Since the time needed to run the application depends on the message size, the horizontal axis was normalized from 0 to 1 representing the start and the end of all the executions. The shaded regions (from 0 to 0.05 and 0.35 to 0.45) mark the transition between the bindings.

3.4 Hubs and Affinity Groups

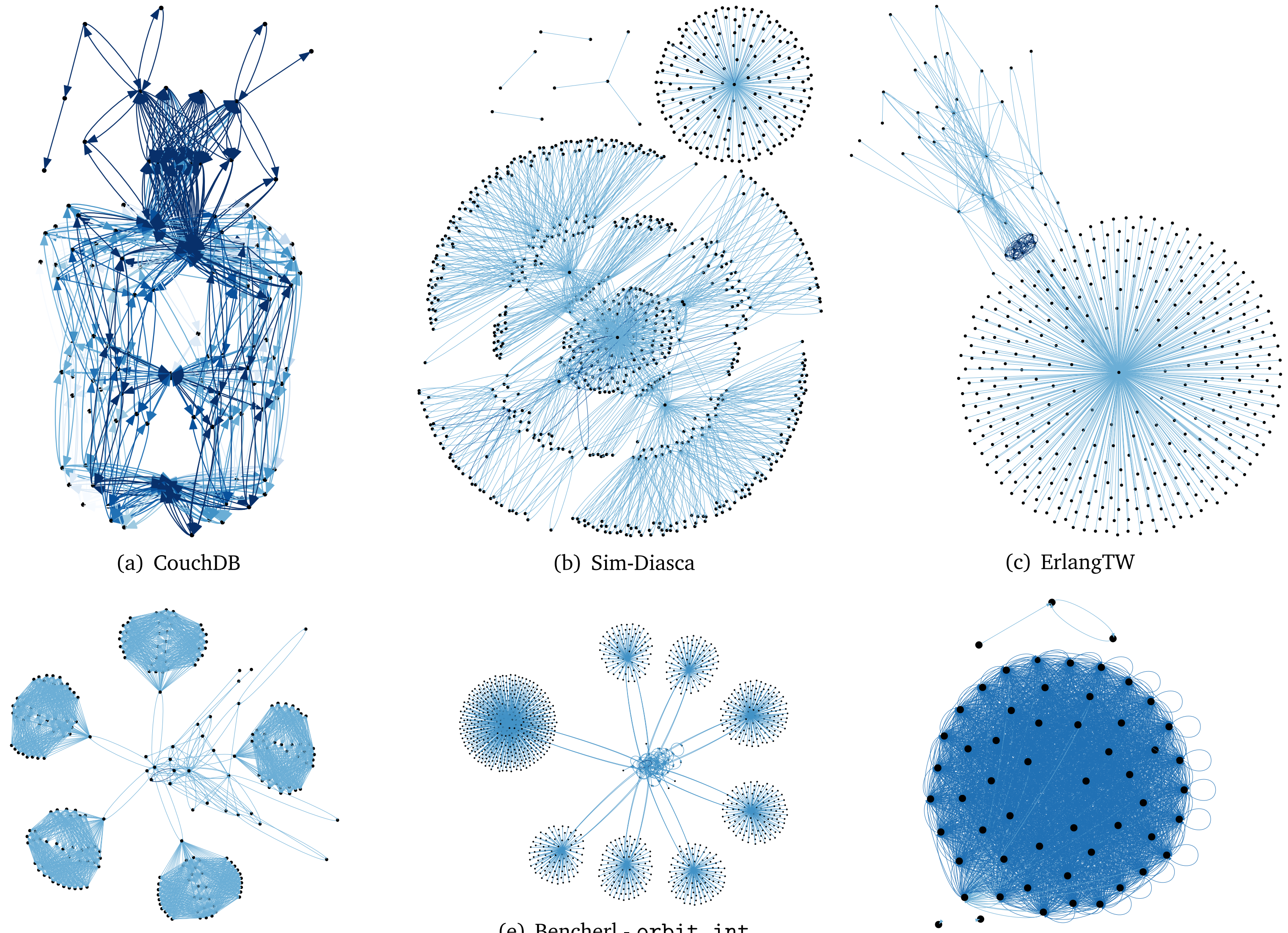
In an actor RE, every function of the application is performed (or at least proxied) by actors that communicate exclusively via message exchanges. We can represent these message exchanges using a graph. In this graph actors are represented by vertices and each individual communication by an edge. In order to graphically represent these graphs we endeavored to produce execution traces for all of the six applications presented in the beginning of the chapter.

To obtain the execution traces we first tried to employ already available Erlang's trace capabilities such as `erlang:trace*` Built In Functions (BIFs), and the dbg, `fprof` and dbg modules [30]. Although quite powerful, these utilities place an unacceptable overhead to the application execution performance. This change in performance, even if it does not change the application output, is sometimes enough to change the communication behavior of the actors. To remedy this problem we chose a lower level solution with better performance: DTrace [GM11]. DTrace is a dynamic tracing framework, originally developed for Solaris, that has a very low performance impact. The Erlang VM already comes prepared to be traced using DTrace and has several pre-defined markers. We used several of these already available markers as well as extended some of them, as for example, to be able to obtain information about the parent of an actor being spawned.

Our experiments were performed on a Linux machine. There are both free and closed DTrace implementations for Linux. Oracle offers a closed solution that only works on *Oracle Unbreakable Enterprise kernel* [31] while the open source solutions we tried were very unstable. Therefore we instead opted for using Systemtap. Systemtap [32] is Linux's equivalent for Solaris' DTrace. Moreover it is also capable of transparently connecting to DTrace markers allowing us to probe the Erlang VM without major code alterations.

The results we obtained are depicted in Figure 3.5. On the top we present the real applications and on the bottom the benchmarks. Every pair of actors that communicated at least once is connected. The number of the communications between each actor pair can vary greatly depending on the involved actors. In order to graphically represent these distinct levels of communication we attributed to each edge a different color. Edges that are displayed in darker colors represent a more intense communication flow while lighter edge tints represent a milder message exchange flow. By analyzing this figure one can realize that each application has quite different communications graphs. As a whole they do not follow a pre-established pattern such as a tree, a ring or a star although these simpler patterns are clearly discernible in some parts of these graphs.

It is also clear that some actors often communicate with a set of other actors while the communication with others is occasional or even non-existent. An actor is created to perform a specific function. Some of these functions are central to the application and are therefore naturally more requested than others. Thus, occasionally, there might exist some actors that are involved in more communications than the average. Employing the graph representation we have just described, this means that the



(a) CouchDB

(b) Sim-Diasca

(c) ErlangTW

(d) Bencherl - ehb

(e) Bencherl - orbit_int

(f) Bencherl - big

Figure 3.5: Communication graphs for three real applications (top) and three synthetic benchmarks (bottom). Vertices represent the actors and edges represent the communications between them. Actors that communicated at least once are connected. The darker the edges the more messages were exchanged.

degree of the vertex representing this actor is higher than the average vertex degree of the graph.

A graph is said to be *scale-free* [BB03] (occasionally referred to as *scale-free network* or *power-law network*) when the probability distribution of vertices' degrees follows a power law. More formally, the probability $P(k)$ that a specific vertex has k edges is given by $P(k) = k^{-\gamma}$, where γ is a graph specific constant. This probability distribution allows for the existence of some vertices with degrees that are sometimes orders of magnitude higher than the average degree of the remaining vertices. In this context, these highly connected vertices are called *hubs*.

Actual examples and the means by which random scale-free graphs are created are still the subject of much discussion [BR04, LADW05]. *Preferential attachment* [BA99] or *Affinity* is one of the most accepted explanations among the proposed mechanisms to explain their creation. This mechanism states that when a new edge is added to the graph the probability it connects to a given vertex is proportional to the current degree of this vertex. In other words, it is probable that highly connected vertices become even more connected while not so well connected vertices remain like that.

Unfortunately, actor communication graphs cannot be described only using this simple set of rules. These graphs are not random and their overall shape or format heavily depend on the specific application. Thus we cannot assert that actor communication graphs are scale-free. However, most application communication graphs present some features that we can also find in scale-free graphs. We will therefore borrow and adapt some of their terminology for use in our own context.

When we analyze communication graphs we realize that some actors are much more connected than others. By the similarity to scale-free graph hubs, we call *hubs* those actors that exchange significantly more messages than the average actor and communicate with a wide variety of distinct actors. Hubs are not only involved with the majority of communications, but also they are responsible by the creation of the majority of actors in an application. Most spawned actors communicate preferentially with their parents or their siblings. We define the set of the actors that exchange messages with a hub actor as the *hub's affinity group*. Figure 3.6 illustrates these definitions using the communication graphs of CouchDB and Sim-Diasca. In this figure some of the hubs and their affinity groups are highlighted.

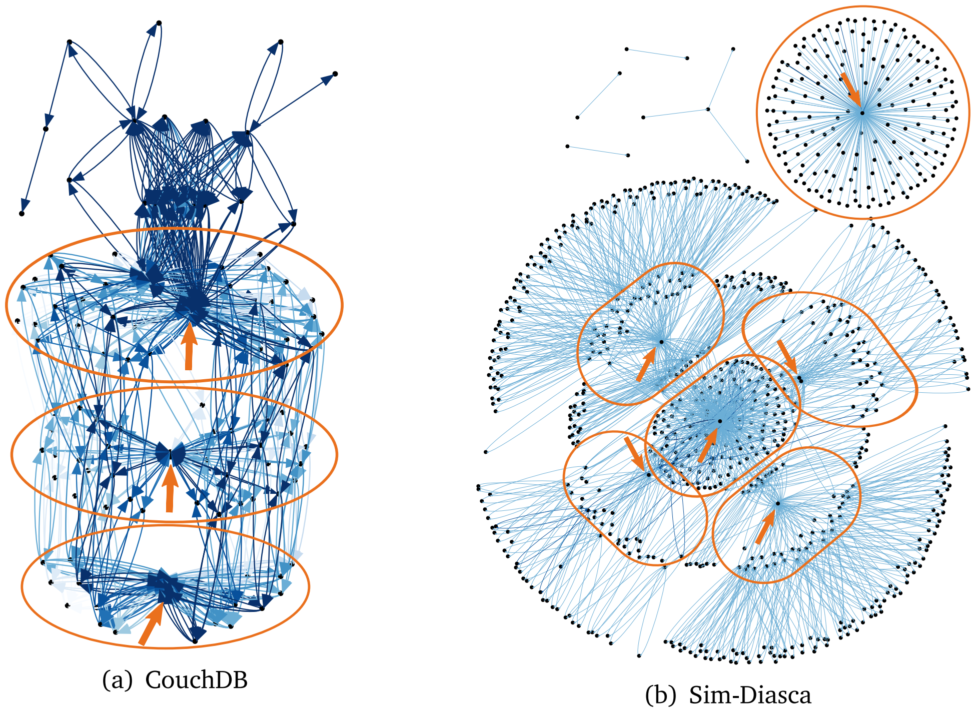


Figure 3.6: CouchDB and Sim-Diasca communication graphs. These are the same graphs depicted in Figure 3.5 however, this time with the hubs and their affinity groups highlighted.

Our affinity-group definition allows for actors to be part of more than one affinity group, that is true even for hubs – a hub may reside in some other hubs' affinity groups. Conversely, each hub *has* only one affinity group. This explains the slight affinity group superpositions visible in Figure 3.6(b).

Similar to actor lifespans and average message sizes, the existence of hubs and their affinity groups is an application trait. Five out of six applications we present possess clearly identifiable hubs. Bencherl's `big` benchmark is the only one that does not. As we previously explained, this benchmark evaluates the performance of the VM on a all-to-all communication scenario. It was constructed so that every actor communicates with every other actor on the system. By our definition this application does not have hubs (which is also the case of the `bang` benchmark that we will see in more details in Chapter 5). But even for those applications that have them, the number of hubs and their behavior (and therefore affinity groups) might also depend on the inputs given to the application. This is the case of virtually all the applications we tested.

Information about the hubs and their affinity groups is available (although sometimes not directly) to the actor RE during the execution of the application. Thus, to define migrations and initial placements the RE could take this information into account.

3.5 Concluding Remarks

We started this section presenting the basic characteristics of actor applications. In order to do that we analyzed three real applications and three applications from a benchmark suite. We showed that an actor lifespan can be very short, that the number of actors in the system is often higher than the number of available PUs and that the communication performance between actors depends not only on the size of the exchanged messages but also on the location of the involved actors.

When we analyzed the behavior of these applications from a higher level perspective, we realized that there are some interesting characteristics that are not being taken into consideration by current REs. We noticed that some actors are involved in more communications than the average actor and that most actors communicate with a reduced number of other actors. We named these highly connected actors *hubs* and the set of actors that frequently communicate with it the *hub's affinity group*.

In the next chapter we will describe how current actor RE running on NUMA platforms can be improved to take these behavioral observations into consideration.

CHAPTER 4

A Hierarchical Approach to Actor Runtime Environments

ACTOR applications are used in a wide variety of scenarios. These scenarios include, web servers, databases, simulators, 3D modeling tools [33], and web development frameworks [Vin12, 34, 35]. The increasing range of actor applications performing server-side roles made the execution of these applications on hierarchical shared-memory platforms very common. Current actor 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. Since NUMA platforms are capable of transparently running code originally created for SMP machines, they have been frequently used as a direct means of performance scaling for current deployments.

Unfortunately some characteristics of NUMA platforms call into question many of the optimization decisions taken by actor RE developers for SMP platforms. 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 architectures, cache-coherence (if available) must be guaranteed by the use of the NUMA interconnections. These NUMA interconnections, as we showed in Chapter 2, might not be a full-graph (in fact they might assume any format) and their performance (in terms of bandwidth and latency) is variable. NUMA platforms present challenges not only to actor REs but also to any concurrent application. The distinct costs to access different parts of memory cause a considerable number of problems that, among others, involve process and memory placement, scheduling, load-balancing, and memory migration.

In Chapter 3 we have shown that actor applications have distinct characteristics such as short lifespans, communication hubs, and communication affinity groups. These features can be used to extract valuable information about the expected behavior of these applications and therefore improve the performance of actor RE on these platforms. Yet, most of these attributes are not being taken into consideration by current actor REs. To the best of our knowledge, no current actor RE considers both the hardware platform characteristics and application knowledge to improve performance on NUMA architectures.

Grounded on the general design choices described in Chapter 2, the knowledge of these hierarchical hardware platforms and the analysis of actor-based applications, we present a behavioral hub-based heuristic. This heuristic's goal is to improve the performance of current actor REs by taking advantage of actor applications' typical features, as well as, the hierarchical nature of the latest shared-memory multi-core platforms. We aim at the performance improvement of actor REs by the reduction of the communication costs imposed by the NUMA interconnections.

In this chapter we review the design choices an actor RE developer has to make with a special focus on the Erlang VM (the basis for our prototype). Section 4.1 presents behavioral heuristic guidelines for actor REs. Then, Section 4.2 presents our proposal for a NUMA-aware actor RE based on knowledge about the application and underlying hardware. We conclude in Section 4.3.

4.1 Heuristic Design Guidelines

In Chapter 3 we presented actor applications and some of their behavioral characteristics. We showed how some of those characteristics, such as lifespan and communication graphs, cannot be predicted for they might depend not only on the application code but also on their inputs. On the other hand, we also described some higher level concepts such as hubs and affinity groups that are present on most applications regardless of the input. We postulate that the existence of hubs and affinity groups in the applications we tested is not a coincidence but rather a common feature of actor application executions.

As we described in Chapter 2, actor applications are used in a wide variety of scenarios. One of those scenarios stands out: highly parallel and scalable systems. Oftentimes the actor model lets application developers create solutions that are much simpler than those created using traditional parallel and concurrent tools. Yet, even in highly parallel systems, there might be some entities that impose limits to the parallelism. An actor-based web-server might have a limited number of connections to the database, despite the number of simultaneous actors serving the requests. Similarly an actor-based database might have countless actors serving or processing requests but, at some point, some synchronization might be needed to consolidate data or to access serial storage devices. I/O actors, however, are not the only culprits. Any serialization point in the application, such as sequential number generators or work queues, will be invariably represented by an actor. This actor will probably become a communication hub since other actors will be actively using its services. One could argue that highly scalable applications will have not one but several hubs otherwise these hubs would quickly become performance bottlenecks.

Yet, we cannot assert that every application presents these features. Although artificial, BenchErl's `big` benchmark does not. This benchmark was created specifically to test Erlang VM's many-to-many communication performance and, because of that, it intentionally does not possess these characteristics. Real applications with no discernible hubs or affinity groups are harder to come by, but they exist nonetheless. In-memory Chord-like DHTs [SMK+01] implemented using the actor model are such an example. This is the reason why we present the following design guidelines for an actor RE as a set of behavioral heuristics[1] rather than a rigid set of rules.

► **Initial Actor Placement** There are several decisions and procedures the actor RE has to perform during the creation of an actor. The first decision is related to the initial placement of this actor. By initial placement we mean the decision of which scheduler, and therefore PU, to which this actor will be assigned. In NUMA platforms this decision also involves the choice of the NUMA node in which the heap of the actor will be allocated.

[1]A behavioral heuristic is a simple rule created based on observation and experimentation that tries to explain the behavior of a system in a simplified manner. By the use of such a heuristic one can anticipate, although not always precisely, the future behavior of this system.

In some applications the expected actor lifespan is very short. Therefore, to provide good performance, the RE needs to have an initial placement policy that is fast. On the other hand, the impact of the initial placement for long-lived actors is not so important. In these cases the RE could take longer to choose a better initial placement. Hubs not only typically live longer than their regular counterparts but also demand a lot more from the RE. Thus, it makes sense to try to spread the hubs in a way that they do not compete for resources. Moreover, regular actors are likely to communicate within their affinity groups, so it makes sense to try to place them close to their hubs. Yet, blindly following these guidelines could lead to severe imbalances among the run-queues. Therefore an actual RE will need to evaluate, in runtime, to what extent they are beneficial to the overall system performance.

► **Affinity Groups** Message exchanges in actor applications are one of most important aspects of the execution. As we showed in the beginning of this chapter, actors tend to communicate preferentially with a reduced set of other actors: its affinity group. It is therefore desirable to keep all the actors of an affinity group as close as possible. In other words, we want to minimize the communication costs by assigning actors to schedulers bound to PUs that are physically as close as possible but, at the same time, try to maintain a good load balance across schedulers, aiming for the best trade-offs in terms of performance.

Even if at the beginning of their lives actors were perfectly distributed throughout the machine, imbalances in the run queues will occur. These imbalances are dealt with by the use of migration algorithms that will try to restore the sizes of the queues to an state where they have approximately the same number of actors. These migration algorithms should take into consideration the hardware characteristics of the underlying architecture as well as the affinity groups to take a decision of which and where each actor should be migrated. On a NUMA machine communication costs between actors might drastically change after a migration. Migrations might involve the migration of the heap to a different NUMA node as well. In this case, not only communication costs but also migration costs might vary depending on the chosen actor's heap size and location.

4.2 A NUMA-Aware Actor Runtime Environment

With the general design guidelines laid down, we are now ready to begin the description of our proposal for a NUMA-aware actor RE. Our approach can be divided in two main aspects: affinity group maintenance and memory management. Fortunately, these two aspects of the execution share a common characteristic. Both of them can be influenced by actor migrations. We will therefore base our approach on the RE mechanisms responsible for the actor placements on the system. These mechanisms can be divided in two complementing categories. The first, initial actor placement, deals with the actor placement during its creation. The second, load-balancing and work-stealing, deals with actor placement during execution. In both cases, however, information about the underlying hardware platform and about the application characteristics are essential for the RE to be able to take better informed decisions.

Limited information about the underlying hardware platform can already be obtained from some actor REs. For instance, in some platforms the Erlang VM is capable of providing a hierarchical view of the machine, including the NUMA nodes. This information is used to bind schedulers to PUs, and therefore concentrate the load of the system on some NUMA nodes during compaction of load cycles. This kind of strategy is however limited. As we have shown, rarely are NUMA interconnections arranged in a linear fashion. Additionally, no information about the number of hops, latency or bandwidth of the different interconnections is taken into consideration, and often the tools needed to obtain this kind of information are proprietary and system dependent. We therefore base our approach on a table provided by the user in which communication costs between PUs are given. The RE does not make any assumption about the meaning of the values other than that bigger values mean higher communication costs.

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. 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 and the heuristic we just defined.

Developers often have good insights into the execution characteristics of the application. Application developers can, therefore, at development time, provide the RE hints about its behavior. These hints include which actors are inclined to create many other actors or to communicate more than the average actor, *i.e.*, hub actors. The goal of these hints is not to change the functional behavior of the applications and the RE can, in fact, at its own discretion, completely disregard them. However, it is our belief that the RE can also use them to help it make better migration decisions. Our approach works by giving the developer tools to flag the actors believed to be hubs.

This flagging 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 of the application. A later decision probably means that it depends on the evaluation of data that is only available during runtime. For example, actors chosen by on-line election algorithms might become hubs during the application execution, thus changing their behavior after their creation and therefore requiring runtime flagging capabilities.

The determination of which actors should be hinted as hubs could also be done using profiling tools after development. The profiling tool we developed to create Figure 3.5 could be used as a visual tool to identify these special actors. Additionally, graph algorithms could be run using execution traces to determine which actors are much more connected. In this thesis we are interested in exploring the execution possibilities when we already know which actors are hubs. However, we also consider the possibility of automatically detecting hubs using graph algorithms in runtime. This idea is further explored in Chapter 7 where we discuss future works.

In the following sections we detail our hierarchical approach to actor REs. First we introduce the concept of schedulers distances, then we outline our NUMA-aware initial placement, load-balancing and work-stealing algorithms.

4.2.1 Schedulers Distance

As we have previously stated, we assume that schedulers are bound to the PUs of the hardware platform. This binding creates, in essence, a direct relation between them. Up until now, we have been talking about the distance between the PUs of a machine and, due to this direct relation between PUs and schedulers, we can also talk about the distance between schedulers. In other words, the communication costs between actors running on schedulers *A* and *B* are proportional to the distance between the PUs to which these schedulers are bound.

The definition of the means by which the distance between the PUs of a machine (and therefore the distance between schedulers) can be classified in at least three categories in increasing order of precision: hierarchical distance, hops-based distance and metric-weighted distance.

The hierarchical distance is measured using the machine's hierarchical memory model. The hierarchical model of a machine can be obtained with tools such as hwloc [BCOM+10]. hwloc is capable of providing a hierarchical tree-like representation of the hardware platform. In this representation the root node corresponds to the machine and the leaves correspond to the PUs. The higher the first common ancestor of two PUs is, the longer is the distance that separates them. Figure 4.1 depicts this representation. In this figure we show communications between the PU #0 of the first processor of the first NUMA node with other PUs showing different communication costs. In some cases the communication can be done through the shared caches (A, B, C) of the processor. The higher the level of the cache the more expensive the communication (A is cheaper than B that is cheaper than C). In other cases (D), the local bus (and eventually the main memory of the NUMA node) may need to be employed. Finally, the most expensive communication (E) involves the use of the NUMA interconnection of the machine.

The hops-based distance category adds a finer level of detail to the distance measurement model. In this category, accesses to memory are not only classified into local or remote but they are also associated to the number of hops needed to access it. The number of hops is the minimum number of NUMA interconnections that must be employed so that two NUMA nodes can communicate. Unfortunately, hwloc fails to represent the interconnection graph between the NUMA nodes and there is no easy portable way to discover this kind of information in runtime since it is frequently locked to vendor specific tools.

Even if such a tool existed, although better than the pure hwloc solution, the hops-based approach still does not correctly reflect the fact that each NUMA interconnection in the machine can present different communication capacities. To remedy this

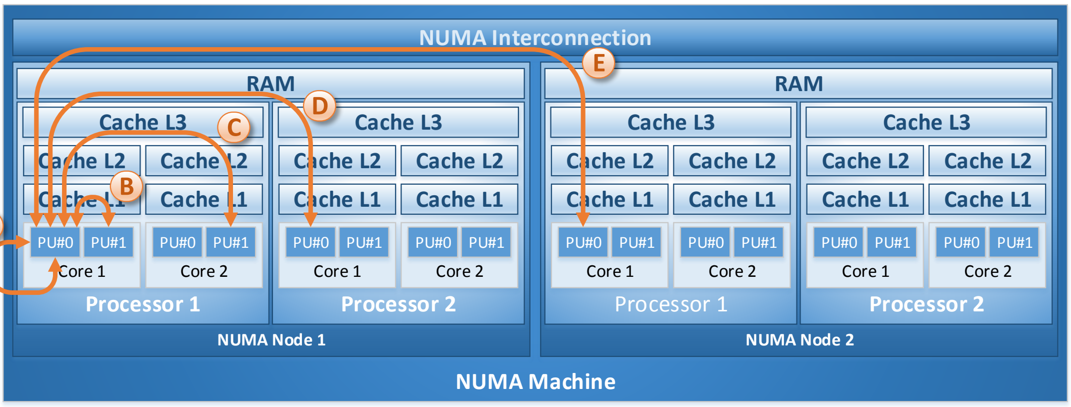


Figure 4.1: Communication costs calculated based on the topology of the machine. In increasing order of cost, from (*A*) to (*E*), the first common ancestor on the memory hierarchy of the PUs involved in the communication are shown. (*A*) shows a local communication. (*B*) and (*C*) show an intra-processor communication using shared caches (L1 and L3 respectively). (*D*) an intra-node communication, in this case the local system bus (and occasionally the node's main memory) is employed. (*E*) an inter-node communication, the most costly, in which the NUMA interconnection is employed.

problem, the metric-weighted distance employs a weighted NUMA interconnection graph. The weights associated to the edges in this graph represent metrics such as latency and bandwidth. The distance between two NUMA nodes is equivalent to the weighted minimum distance between the nodes on this graph. HieSchella [PRC+14], an extension to hwloc, is capable to produce such a graph. In fact, to maintain interface compatibility and due to memory consumption reasons, internally Hieschella continues to use a tree to represent the platform topology. Therefore, to adequately provide distance metrics between arbitrary NUMA nodes, each node of the tree that represents a NUMA node has additional data fields that hold vectors with the communication costs to every other NUMA node on the system [36]. To calculate these communication costs in a portable fashion it uses some known benchmarks such as LMbench [MS+96] and coNCePTuaL [Pak07] for memory and inter-node bandwidth and latency evaluations.

For the remainder of this chapter we will abstract the details of how the actual communication costs are calculated and assume that a function Cost(i, j) with the communication costs between schedulers i and j is provided for each platform. Let n

be the number of schedulers (PUs) available to the RE. Then this function will be such that $\mathrm{Cost}(i,i) \leq \mathrm{Cost}(j,k), \forall i,j,k \in [1,n]$ (local communications are cheaper than non-local).

The costs determined by this function, as we will see further on, are only used as a numeric value to determine the distance between two schedulers and are not assumed to possess any unit. Treating this function as dimensionless has the advantage of giving the end-user the liberty to feed the RE (using for example a text file) any metric they see fit. Such metrics include, but are not limited to, the NUMA links bandwidths, latencies or the number of hops between PUs. It also gives the user the opportunity to experiment with different values that do not directly follow the hardware platform characteristics to fine-tune the application execution. For instance, applications which have frequent message exchanges of small sizes might want to define communication latency as their primary metric, whereas applications that exchange larger messages, but with a lower frequency, might want to prioritize bandwidth.

During the application execution, imbalances are bound to happen even with a good initial placement policy. That is the reason why the RE needs actor migration mechanisms. However, to improve the overall performance of the system on NUMA architectures, the way a candidate is chosen for migration matters. Current REs migrate actors mainly based on the size of the queues. What we propose is that the topological distance between the queues, more specifically, the distance between the PU to which the scheduler that is owner of the queue is bound, also be taken into consideration. That is, actor migrations would be primarily done between queues that are near and only then between queues that are farther. In order to do that, the RE needs some runtime support which we now explain.

4.2.1.1 Runtime support

To provide the actor RE with the necessary support to determine the distance between schedulers, we will employ the Cost function we just defined. During the RE initialization each scheduler will calculate the distances between itself and every remaining scheduler and then store them in a vector. This vector (`dist`) will keep the indices of these schedulers in non-decreasing order of distance. The scheduler itself is not included in this vector. Figure 4.2 shows a hypothetical example of the function Cost and the `dist` vectors for a RE with three schedulers.

PU From	PU To	Cost
1	1	1.0
1	2	1.5
1	3	2.0
2	1	1.5
2	2	1.0
2	3	1.5
3	1	2.0
3	2	1.5
3	3	1.0

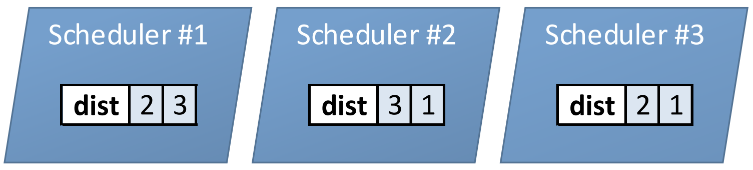


Figure 4.2: Cost function (here represented as a table) and distance vector associated to each scheduler. During the initialization each scheduler initializes the vector using the user-provided Cost function.

Similar to the `dist` vector, the RE will also calculate during its initialization the average distances between every NUMA node pair. Thus each NUMA node will keep a list (`numaNodesInDistanceOrder`) in non-decreasing order of distance with the indices of the remaining NUMA nodes. By default, the average distances between NUMA nodes are based on the Cost function. Let $scheds_i$ be the set containing all the scheduler indices of the i-th NUMA node. Then the average distance between NUMA nodes i and j is given by:

$$numaDist(i,j) = \frac{\sum\limits_{k \in scheds_i} \sum\limits_{l \in scheds_j} \mathrm{Cost}(k,l)}{|scheds_i| \times |scheds_j|} \tag{4.1}$$

Similar to the *Cost* function, the actual *numaDist* function can be arbitrarily chosen. Our approach does not have any restriction to its form other than that remote communications should be at least as costly as local communications. That is, $numaDist(i,i) \leq \mathrm{Cost}(j,k), \forall i,j,k$.

With the `dist` and `numaNodesInDistanceOrder` vectors pre-calculated, distance-aware migration mechanisms can search schedulers and NUMA nodes for migration candidates just by following the order in which they appear in these lists.

4.2.2 Initial Actor Placement

There are several possible policies to place a newly spawned actor depending on its expected behavior. Proportionally, hubs demand more resources from the RE than their regular counterparts and are usually 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, therefore providing a better initial distribution of load and conceivably improving the affinity group maintenance.

In fact, the best way to spread hubs will depend on the application behavior. For example, we could privilege communication by placing hubs close but not on the same PU, or privilege resource independence by placing actors as far as possible. Both these strategies promote a good initial distribution of hubs among the available PUs. We propose the following possible initial placement strategies:

- *Default* – Places actors on the same scheduler of their parent, it is also the default (and currently the only option) on the Erlang VM.
- *Compact* – Favors communication between actors by placing them close together.
- *Scatter* – Privileges resource independence by placing actors as far as possible from each other.
- *Circular* – Performs a round-robin algorithm to distribute the spawned actors.
- *Random* – Chooses a random scheduler.

We modified Erlang VM to provide support for these five policies. They can be defined in runtime and it is possible to define distinct policies for regular and hub actors.

Even if the initial placement policy could deploy actors using an optimal placement, imbalances are bound to happen during the execution of the application. For this reason the actor RE also counts with migration mechanisms that are responsible to restore the balance between the schedulers. We discuss these mechanisms in the next sections.

4.2.3 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 should try to keep actors, and their affinity group, close together so that communication between them is fast. Sometimes these two goals may conflict. For example, 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 first introduce how current mechanisms for load-balancing work on the current Erlang VM and then we propose our set of modifications

4.2.3.1 Erlang VM's Load-balancing Mechanisms

Erlang's VM load-balancing mechanism works by trying to keep every scheduler run-queue with approximately the same number of actors. For that it first defines how many actors each queue should contain and then establishes a target number of actors, the migration-limit, for each one of these queues. This target value is calculated independently for each queue priority. Next, it defines migration paths. Migration paths determine from and to which schedulers actors should be migrated.

The load-balancer routine is activated at regular intervals. These intervals are determined by the number of reductions[2] executed by the schedulers. The first scheduler to reach a pre-established reduction limit becomes the responsible for the execution of the load-balancer. During these intervals, statistics about each one of the scheduler run-queues are gathered to calculate the target number of actors per run-queue (migration limit).

Erlang VM's calculation of the migration limit is quite intricate and it is not essential for the understanding of the remaining of this chapter. As a matter of

[2]A reduction is roughly one function call.

fact, our approach does not depend on the actual function used to obtain this value. Therefore, we provide a full explanation on how the Erlang VM calculates migration limits and migration paths in Appendix B. Without loss of generality, for the remaining of this chapter, the migration limit can be considered as being the average number of actors in each run-queue during the last interval between the load-balancing rounds. While fully understanding migration limits calculation is not necessary, we will need at least a rudimentary understanding of the way migration paths are chosen.

In order to calculate migration paths, the RE gathers usage statistics for each scheduler (and its run-queues) during the execution. This information is used to determine if it is underloaded or overloaded in comparison to the remaining schedulers. The load evaluation is based on historical number of reductions executed by each scheduler in relation to the whole RE. If the load of a scheduler is below the target value (the migration limit) then it becomes a destination for actor immigration. On the other hand, if the load is above the target value, it becomes a source of actor emigration. Emigration and immigration flags are set in each run-queue to indicate in which case each run-queue is. In the following step schedulers are ordered (independently for each actor priority) in non-decreasing order of load and migration paths are set. Basically the paths are set from the most overloaded scheduler to the most underloaded scheduler, then from the second most overloaded to the second most underloaded, and so on. Some other simple rules exist to deal with mismatched number of underloaded and overloaded schedulers.

For the sake of simplicity, from now on we assume that only the default actor priority is used (this is in fact the usual case). This assumption does not cause any loss of generality since the mechanisms that control the balance of the remaining priorities are independent and the same. Therefore, the proposed solution could be directly applied for any run-queue regardless of its priority.

4.2.3.2 A Hierarchical Approach

Like Erlang VM's load-balancing algorithm, our approach is also based on the run-queue sizes. We believe that the actual migration limit calculation (needed to keep Erlang's soft real-time properties) is not the problem, but rather the way in which the candidate actors are chosen for migration. Migration paths are set taking into account only the load of each scheduler. The distance between these schedulers is not considered, so even if a few local (inside the same NUMA node) migrations could solve the imbalance, actors migrations across NUMA nodes are very likely to happen. Additionally, even if the choice of schedulers were done optimally, the choice of the actor to be migrated is currently arbitrary: the VM just picks the actor that happens to be the first on the target run-queue.

Our hierarchical approach to load-balancing also employs migration limits and migration paths. There are however two major differences in our load-balancing algorithm. The first difference is the introduction of local load-balancing rounds. The second difference is the algorithm used to choose the candidate for migration.

The local load-balancing algorithm works almost the same way as the Erlang VM's algorithm. The difference is that the migration limits and the migration paths are only calculated for the schedulers bound to that specific NUMA node, disregarding the information collected by schedulers bound to PUs in other NUMA nodes. Every NUMA node will then independently execute its own local load-balancing. The idea is that by avoiding inter-node migrations we can improve the overall efficiency of the system by, for example, promoting better use of the processor caches and the NUMA interconnections.

However, the exclusive use of the local load-balancer strategy might create imbalances between the NUMA nodes. For that reason, the average load of each NUMA node is calculated at the beginning of each local load-balancing iteration. If the average load of the NUMA nodes differs by more than a parameterized threshold a global load-balancing round is started. The global load-balancer also works with migration limits and migration paths, however, it regards all the run-queues of NUMA node as if they were only one. Migration limits and migration paths are calculated for and between whole NUMA nodes and not their schedulers. With this approach we can guarantee that the balance is maintained across the whole RE.

Cross-node migrations not only are more expensive than local migrations but also the migrated actor will probably (*cf.* Chapter 3) become more distant from its hub and affinity group. As a matter of fact, migrations only change the execution placement of the actor. Its heap, therefore its home node, continues unaltered hurting its execution performance since it has to perform remote memory accesses.

At this point, we need to choose what to do with the heap of a migrated actor.

There are basically two possible choices. The first, as we explained, is to leave the actor heap (and mailbox, and stack, and every other data structure used exclusively by it) where they are. Eventually, in the new location the actor might need to grow (or shrink the heap after a garbage collection) and when new heap is allocated it will be done in the new location and its home-node will be altered. Until then, access to these data will be remote. The second option is to migrate all the actor's data structures to the new location. This way, all the new accesses will be local and therefore fast.

We believe the first option, *i.e.*, not to migrate the data structures, is the most appropriate. Actors tend to live short lives. The time needed to perform the migration of the heap might quickly become an important performance impacting factor if the migrations are too frequent or the heap is too big. Additionally, our approach tries to keep actors near their data structures, *i.e.*, their home nodes. The behavioral heuristic tells us that by doing so we will be increasing the affinity between the actors. Thus, our proposed load-balancing mechanisms primarily try to bring processes home before any other actor migration is attempted. Eventually, some actors might be migrated to a different NUMA node, however at the first opportunity they will be brought back home. Moreover, if during this interval of time the heap of the actor is migrated (after a heap growth for example) we will privilege the migration of other actors that are away from home before forcing this actor to be migrated to another node, once again, away from home. These migration strategies have no impact on SMP platforms but on NUMA machines they have the potential to significantly change the performance of the RE.

Our approach to load-balancing has two distinct phases. In the first phase, the decision of the load-balancing type (local or global) is made, and the migration limits and migration paths are calculated. In the second phase, the scheduler loop performs the actual actor migrations before scheduling any actor for execution. The scheduler loop is also responsible for the decision of calling the load-balancer parameters calculation after a pre-defined elapsed interval. We now present how these two phases work in more detail.

Phase One - Load-balancer Parameters Calculation This phase of the load-balancing algorithm does not migrate actors by itself. This is done at the beginning of each scheduling cycle that follows the parameters laid out by this phase. These parameters include migration limits and migration paths for each scheduler.

Algorithm 4.1 shows its implementation sketch. First it verifies if there are not any other load-balancing routines running at the same time (Lines 3-5). For that if checks the `localLoadBalancerRunning` and `globalLoadBalancerRunning` flags. `localLoadBalancerRunning` is actually an indexed array, that keeps the flags for each NUMA node independently. If there is a current execution of the procedure in the local NUMA node or a global execution the procedure simply exits. That is the reason why Lines 3-12 are in an atomic block. This ensures that the view of the system is consistent across schedulers in the same and on distinct NUMA nodes. Additionally, since the calls for the `CalculateLoadBalancingParameters` procedure can happen at the same time independently for two distinct schedulers on the same NUMA node, the number of reductions for each scheduler is verified once again inside the atomic block (Line 4). The number of reductions executed by each scheduler is used as a trigger to the execution of the load-balancer.

Next, the algorithm determines if a global or local load-balancing will be performed (Lines 6-12). For that, it calculates the migration limits for each NUMA node and if the ratio of the difference between the minimum and the maximum migration limits and the minimum migration limit is higher than a threshold, then a global load-balancing round is started.

In both cases, global or local, the following step (Lines 13-31) is to calculate the migration paths and reset the schedulers reduction counters. If it is a global load-balancing round then all counters from all schedulers are reset whereas if it is local just the counters from the schedulers in the local node are reset. In the local case there is still an extra step in which the migration limits for every local scheduler are calculated (for the global case these limits had already been calculated in Line 7).

Phase Two - Scheduler Loop Similar to the original Erlang VM, in our proposal each scheduler works using a loop in which it continuously looks for tasks and executes them. Algorithm 4.2 brings its implementation sketch in pseudocode. In the first part (Lines 3-16) it verifies if migration flags have been set and if true, depending if it is a global or a local load-balancing round, it chooses the actor to be migrated following the migration paths that were previously established. In the

Algorithm 4.1: Load-balancer parameters calculation procedure sketch

```
1  Procedure CalculateLoadBalancingParameters
2      localNode ⟵ currentScheduler.numaNode;
3      atomic
4          if localLoadBalancerRunning[localNode] or globalLoadBalancerRunning or
           currrentScheduler.reductionCounter < minimumReductionsForLoadBalancer then
5              Exit;
6          else
7              nodeLimits ⟵ CalculateNUMANodesMigrationLimits();
8              diff ⟵ (Max(nodeLimits) - Min(nodeLimits)) / Min(nodeLimits);
9              if diff > threshold then
10                 globalLoadBalancerRunning ⟵ true;
11             else
12                 localLoadBalancerRunning[localNode] ⟵ true;
13     if globalLoadBalancerRunning then
14         CalculateNUMANodesMigrationPaths(nodeLimits);
15         schedulers ⟵ GetAllSchedulers();
16         atomic
17             for s ∈ schedulers do
18                 s.reductionCounter ⟵ 0;
19             Set global migration flags;
20             globalLoadBalancerRunning ⟵ false;
21     else
22         localQueues ⟵ GetAllLocalQueues();
23         for q ∈ localQueues do
24             queueLimits(q) ⟵ CalculateMigrationLimit(q);
25         CalculateMigrationPaths(localQueues, queueLimits);
26         atomic
27             localSchedulers ⟵ GetLocalSchedulers();
28             for s ∈ localSchedulers do
29                 s.reductionCounter ⟵ 0;
30             Set local migration flags;
31             localLoadBalancerRunning[localNode] ⟵ false;
```

local case, it does not matter which actor we choose. So we just take the head of the run-queue. For the global case, we use an auxiliary function to make this choice (`ChooseBestActorFromNode`, described in Algorithm 4.3). If, on the other hand, the migration flags are clean, then the scheduler verifies if its current number of reductions is enough to start a new load-balancing round (Lines 15-16) and calls the `CalculateLoadBalancingParameters` function which we just described.

Algorithm 4.2: Scheduler loop implementation sketch

```
1  Procedure SchedulerLoop
2      while true do
3          if currentScheduler.migrationFlags are set then
4              if migration flagged as local then
5                  runQueue ⟵ follow migration path to a local run-queue;
6                  actor ⟵ Head(runQueue);
7                  if runQueue.migrationLimit was reached or actor $= null$ then
8                      Clean currentScheduler.migrationFlags;
9              else
10                 numaNode ⟵ follow migration path to a remote NUMA node;
11                 actor ⟵ ChooseBestActorFromNode(numaNode,
                   currentScheduler.numaNode);
12                 if numaNode.migrationLimit was reached or actor $= null$ then
13                     Clean currentScheduler.migrationFlags;
14             Migrate(actor, actor.runQueue, currentScheduler.runQueue);
15         else if currrentScheduler.reductionCounter $\geq$
           minimumReductionsForLoadBalancer then
16             CalculateLoadBalancingParameters();
17         else if currentScheduler.runQueue $\neq \emptyset$ then
18             actor ⟵ Head(currentScheduler.runQueue);
19             actualReductions ⟵ ExecuteActor(actor, maxNumberOfReductions);
20             currentScheduler.reductionCounter ⟵
               currentScheduler.reductionCounter + actualReductions;
21             if actor.isActive then
22                 InsertTail(currentScheduler.runQueue, actor);
23         else
24             WorkStealing();
```

In the following step, the scheduler passes to the execution of an actor (Lines 17-24). For that, it first takes out the first actor in the queue (Line 18) and executes it until either it exhausted its number of reductions (defined by the `maxNumberOfReductions` parameter) and is preempted by the RE or the actor itself stops its execution (Line 19). An actor can voluntarily stop executing if it has finished or, for example, it is waiting for a message. In this case the number of reductions will be inferior to the maximum. Line 20 accounts the actual number of executed reductions and then, if the actor is still active, it is put back to the end of the run-queue to wait for the next execution slot (Lines 21-22). If, by any chance there is no actor to be executed, *i.e.*, the run-queue is empty, the scheduler will try to steal actors from other schedulers (Line 24). We will cover this case in Section 4.2.4.

In order to present the actor choice algorithm, we first need to introduce the concept of a *foreign actor*. A foreign actor is simply an actor whose home node is not the same NUMA node to which its current scheduler is bound. This may happen as a result of a migration. Each time an actor is migrated, the destination scheduler checks if its NUMA node and the actor's home node are the same and if not it adds the actor to a list of foreign actors. The lists of foreign actors can be quickly searched by NUMA node identifiers and each scheduler keeps its own list of foreign actors for which it is responsible. Whenever an actor is migrated, theses lists are updated. Additionally, during initialization, not only each scheduler initializes a list which contains the remaining schedulers in non-decreasing order of distance, but also the RE also calculates and stores, for each NUMA node, a list of the remaining nodes in non-decreasing order of distance.

This process is described by Algorithm 4.3. When an actor needs to be chosen for migration from a remote NUMA node, the RE will first try to bring actors home, *i.e.*, it will look up the foreign processes lists on the remote node for its own NUMA node and check if there is an actor that can be brought home. This has the effect of bringing actors closer to their heap and hopefully closer to their affinity group and, therefore, can improve overall communication and execution performance (Lines 2-4).

If no actor in this condition is found, it will then try to migrate the remaining foreign actors in non-decreasing order of distance (Lines 5-9). The rationale here is that a foreign actor in the remote NUMA node already is not as fast as it could be, therefore migrating a foreign actor could minimize performance losses. The order in which we choose the actor for migration has the same motivation, *i.e.*, during execution the migrated actor will be the one that minimizes the new distance to its

Algorithm 4.3: Actor choice algorithm sketch

```
1  Function ChooseBestActorFromNode (fromNumaNode, toNumaNode)
2      foreignList ⟵ fromNumaNode.foreignActorList[toNumaNode];
3      if foreignList $\neq \emptyset$ then
4          Return Head(foreignList);
5      otherNodes ⟵ toNumaNode.numaNodesInDistanceOrder;
6      foreach node in the list otherNodes do
7          foreignList ⟵ fromNumaNode.foreignActorList[node];
8          if foreignList $\neq \emptyset$ then
9              Return Head(foreignList);
10     runQueue ⟵ Select the run-queue with maximum length from
       fromNumaNode.runQueues;
11     actor ⟵ findAndRemoveFirstNon-HubFrom(runQueue);
12     if actor = null then
13         actor ⟵ Head(runQueue);
14     Return actor;
```

home node. Finally, if no foreign actor is present on the remote NUMA node, then the first actor on the remote node in the run-queue with the maximum length is chosen (Lines 10-14).

4.2.4 Work-Stealing

Between the load-balancing rounds, some schedulers might run out of work. Instead of waiting until these schedulers' run-queues are replenished by the load-balancing algorithm we just presented, a temporary lightweight solution is employed. This solution is called work-stealing. Work-stealing is a procedure in which actors are "stolen" (migrated) by idle schedulers from the remaining schedulers. This guarantees that, if there is enough work, every scheduler is kept busy until a more permanent load-balancing solution can be employed.

4.2.4.1 Erlang VM's Work-Stealing Mechanism

In order to improve communication performance in underloaded systems and reduce how often the schedulers run out of work, the Erlang VM might decide to *suspend* some schedulers in a strategy called *compaction of load*. This strategy is enabled by default and can be used to improve actor affinity or resource independence. It works by suspending schedulers when the charge of the system is not high enough. The schedulers chosen for suspension are always those with the highest identifiers. With less active schedulers for the same amount of actors, 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 actors between schedulers. Moreover, since suspended schedulers are those with the highest identifiers, given that appropriate scheduler bindings are used, compaction of load can be used to improve actor affinity on a NUMA architectures by, for example, migrating actors to a smaller set of schedulers 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 linear view of the NUMA architecture. Usually NUMA nodes are not organized in a linear fashion, limiting the impact of this strategy.

The current strategy of the Erlang VM consists of first trying to steal actors from suspended schedulers (to avoid stalling their execution until the next load-balancing round). If unsuccessful, the algorithm tries to steal from the remaining ones. The candidate schedulers are evaluated on ascending identifier order starting from the stealer scheduler 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 $i+1$, $i+2$, $\dots$, n, 1, 2, $\dots$, $i-1$.

This stealing strategy suffers from the same problem of the compaction of load. The migration choices can be made in a way to respect the NUMA interconnection up to a point, however it still suffers from an artificial linear arrangement of schedulers distance.

4.2.4.2 A Hierarchical Approach

Similar to what we did to accomplish a hierarchical load-balancing approach, we will also use the communication costs information provided by the Cost function. The idea is similar to that we used to choose which actor should be migrated on the load-balancing case. The sketch of this process is outlined by Algorithm 4.4.

Algorithm 4.4: Hierarchical work-stealing algorithm sketch

```
 1 Procedure WorkStealing
 2     localNode = currentScheduler.numaNode;
 3     schedulersDistances = currentScheduler.dist;
 4     foreach scheduler in the list ReverseOrder(schedulersDistances) do
 5         foreignList = scheduler.foreignActorList[localNode];
 6         if foreignList $\neq \emptyset$ then
 7             actor = Head(foreignList);
 8             Migrate(actor, actor.runQueue, currentScheduler.runQueue);
 9             Exit;
10     localSchedulers = localNode.schedulers;
11     foreach scheduler in the list schedulersDistances such that scheduler $\in$
       localSchedulers do
12         runQueue = scheduler.runQueue;
13         if runQueue $\neq \emptyset$ then
14             actor = Head(runQueue);
15             Migrate(actor, runQueue, currentScheduler.runQueue);
16             Exit;
17     otherNodes $\longleftarrow$ localNode.numaNodesInDistanceOrder;
18     foreach node in the list otherNodes do
19         actor = ChooseBestActorFromNode(node, localNode);
20         if actor $\neq$ null then
21             Migrate(actor, actor.runQueue, currentScheduler.runQueue);
22             Exit;
```

This algorithm employs three different steps, in order of preference, to try to find a candidate actor to be stolen.

The first step (Lines 2-9) consists in bringing actors back home. It does so in non-increasing order of distance, *i.e.*, it first tries to bring actors that are farther away before bringing actors that are nearer. For that it uses the distance vector we described in Section 4.2.1. The rationale of this first step is that by bringing back foreign actors, performance can be improved. The farther they are the better the expected performance gains since we will be improving communication locality.

There might not be any actor in this situation and, in this case, a second technique is applied (Lines 10-16). The idea is to keep the impact of the migration to a minimum, for this reason only local schedulers are considered in this step. They are nonetheless evaluated in non-decreasing order of distance, that is, even in the same NUMA node we try to steal first from schedulers that are closer and only then from schedulers that are farther.

If the previous two steps are fruitless, the RE has no option other than try to steal actors from the schedulers on the remaining NUMA nodes (Lines 17-22). To minimize the impact of this migration it also does so by non-decreasing order of (NUMA) distance and uses the *ChooseBestActorFromNode* function we defined for the hierarchical load-balancing algorithm (Line 19).

4.3 Concluding Remarks

With this chapter, we conclude our description of the actor model, SMP and NUMA architectures, the inner workings of current actor REs, actor application behaviors and our proposal for a NUMA-aware RE. We have shown that, although highly optimized for SMP machines, current actor REs lack specific runtime optimizations to better suit the needs of these hierarchical platforms.

Taking into consideration the behavior we observed, we proposed a set of performance improvement guidelines for actor REs. These guidelines are based on a simple heuristic intended to explain the behavior of common actor applications. As a heuristic-based approach, it might not be suitable to some actor applications and its applicability might need to be evaluated on an individual basis.

In addition to this heuristic, our optimizations make use of hints provided by the application developer. This course of action was chosen so that we could minimize the overhead imposed on the system. We presented new concepts and approaches that, to the best of our knowledge, are original in actor REs such as:

- Platform and hub-aware initial placement policies.
- Home-nodes and foreign actors tracking.
- Hierarchical migration (with the “bring home” strategy) for both load-balancing and work-stealing.

To assess how our proposal fares in practice, in the next chapter we evaluate the performance of our prototype based on a real actor RE running on NUMA platforms.

CHAPTER 5

Experimental Evaluation

HIGHLY optimized actor runtime environments for SMP architectures already are the norm. On the other hand, despite the fact that hierarchical shared memory multi-core platforms are present in many mission-critical actor-based systems, to the best of our knowledge, current RE optimizations that deal with the distinctiveness of NUMA platforms are very limited. Taking into consideration the observations presented in Chapter 3 and our proposal for a hierarchical approach to actor RE described in Chapter 4, we modified the codebase of a popular and real actor RE, the Erlang VM. In this chapter we present the realization and experimental evaluation of our proposal for a NUMA-aware actor RE.

In the first part, we characterize our evaluation methodology. We describe the modifications we introduced to the Erlang VM codebase – necessary to evaluate our proposal. Then, we outline the experimental platform used during our experiments. Next, we present the experimental results. We evaluated the performance of the modified VM against real applications and well known benchmarks and show that, if the application characteristics fit our assumptions of the problem domain, we can indeed create a RE with better performance than current actor REs provide.

The remaining of this chapter is organized as follows. Section 5.1 describes our evaluation methodology. Then, Section 5.2, presents our experimental results. Finally, we conclude in Section 5.3.

5.1 Evaluation Methodology

In this section we present all the software and hardware frameworks used for the evaluation of our proposal. We begin by outlining the modifications we did to the Erlang VM to then present the hardware platform used during our tests.

5.1.1 The Modified Erlang VM

The Erlang VM has several interesting run-time tools that can be used to tune and profile applications, even on NUMA platforms. However, to perform the analysis we present in this document and to realize our proposal for a NUMA-aware actor RE, we needed to adapt and extend the current Erlang VM. In this section we briefly describe these modifications[1].

We have used Erlang R15B02 code base as the starting point for our NUMA-aware VM. For the remaining of this text we will refer to the pristine Erlang VM as *original* and to our own custom VM as *modified*. 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. Therefore all of our tests were performed on this OS. In spite of that, 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.

5.1.1.1 Hubs Tracking

An important part of our proposal involves the tagging of hub actors. On the VM code level, an actor is represented by a struct and the list of the alive actors is kept by the VM as a plain C array. The size of this array determines the maximum (parameterizable) number of actors in the system. When an actor is created, an available slot on the array is claimed and filled with the pointer to it. 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 actors using

[1]We believe that the explanations we have given so far about the inner workings of the Erlang VM are enough for the full understanding of this section. However, more thorough explanations about the current VM implementation are provided by Zhang [Zha11] and by Stenman [Ste02]

an intrusive linked-list[2] in the actor 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 actor 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 actors as hubs. These markings take the form of hints. These hints can be given as an extra parameter during the actor creation or by setting a flag of a running actor. 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 actors 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 actors to a specific scheduler.

The interface we created to allow application developers to flag hubs is shown in Listing 5.1. As a basis for comparison, Lines 2-3 show the creation of a regular actor. Lines 5-6 show the creation of a hub actor using the new interface. Finally, Line 8 shows how to flag an existing actor as a hub and Line 10 shows how to remove this flag.

```
 1  % Regular actor
 2  Pid1 = spawn_opt(A_Module, A_Function, FunctionArgs,
 3      []).
 4  % Hub actor
 5  Pid2 = spawn_opt(A_Module, A_Function, FunctionArgs,
 6      [hub_process]).
 7  % Flags calling actor as a hub
 8  erlang:system_flag(hub_process, true).
 9  % Removes hub flag from the calling actor
10  erlang:system_flag(hub_process, false).
```

Listing 5.1: Erlang interfaces to create and set hub actors

[2]An intrusive linked-list is a linked-list whose linking pointers are allocated inside the object it holds. This allows not only insertion, but also removal and membership tests in $\mathcal{O}(1)$ time.

5.1.1.2 Actor Migration Policies

We have adapted the proposed hierarchical approach presented in Chapter 4 to the Erlang VM. We modified the Erlang's actor-choice algorithm to take into consideration both the architecture of the machine as well as the home node of each actor.

Scheduler distances are calculated at initialization time with a user provided Cost function or, by default, using as cost function the hierarchical distance between the PUs. Due to our modifications to the migration mechanisms, as long as schedulers are bound to the PUs, the actual binding choice makes no performance difference, *i.e.*, the calculated distance vectors will always reflect the binding choice in use by the RE.

The behavior of both the load-balancer and the work stealing algorithms can be defined or queried using the `erlang:system_flag/2` and `erlang:system_-info/1` function calls with the atoms `scheduler_migration_strategy` and `scheduler_ ws_strategy` respectively. The available behaviors for both algorithms are `default`, `disabled`, and `numa`. The `disabled` strategy completely disables the work-stealing or load-balancing algorithm and, since it decreases the overall system performance, it is useful only for debugging purposes.

No information about the topology of the machine is ever considered by the current load-balancer. 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 deals with the choice of the actor to be migrated. This ensures that we will first try to bring actors home, then to migrate actors inside the same NUMA node and only then between NUMA nodes.

Similar to Erlang's VM 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 actors home, *i.e.*, it will try to find a actor for which the home node is the same of the stealing scheduler. If no process fulfills this criterion, it will try to steal any actor. If no actor is found during this step, the algorithm will try to steal from the active schedulers. Like the previous step, it will first try to bring actors home and only then consider the remaining actors. Another important difference from the default strategy is the order in which the schedulers are scanned. By default, schedulers are scanned sequentially.

In our approach, they are scanned in non-decreasing order of distance as defined by the provided cost function (*cf.* Chapter 4). As a direct consequence, it will search schedulers on the same node (therefore not changing the actor home node) and only then consider the remaining schedulers.

Due to the proposed migration algorithms, actor home-nodes and foreign actors also need to be tracked. To track each actor home node, we just use an additional member in the actor data structure to keep this information. It is important to note that this location is not constant throughout the application execution. For example, after a heap growth operation, the new heap location will be the same as its current execution location, which is different from its home node in the case of foreign actors. For this reason, each scheduler's list of foreign actors needs to be kept in an efficient data structure that allows fast updates. To that end we will employ, once again, intrusive lists. In this case however, for a platform with n schedulers, each scheduler will keep a pointer to the head of $n-1$ intrusive lists, one for each remaining scheduler. The actual linked-list nodes are stored inside the actor data structure itself. Whenever an actor is migrated to a new NUMA node, a heap relocation happens due to heap growths or garbage collections, or an actor dies, these lists can be updated accordingly in constant time. Additionally, load-balancing mechanisms can enumerate all the foreign actors on schedulers of remote NUMA nodes in an efficient way.

The actual implementation of this modification in the Erlang VM employs the *Strategy* design pattern [JHVG95], allowing an easy creation of new migration policies. Thus, it is now simple to add new work-stealing and load-balancing algorithms if desired. The requirements are the implementation of a new function and the addition of a pointer to this function to the list of available strategies. Like the initial placement strategy we will see in the following section, this strategy can be changed in run-time.

5.1.1.3 Initial Placement Policies

The default policy of the Erlang VM is to place each newly spawned actor on the same scheduler of its parent, therefore, expediting its creation specially due to the memory allocation and heap initialization. We modified the Erlang VM to introduce some new initial placement policies. As we previously mentioned, our implementation employs the *Strategy* design pattern. To create new initial placement choices, it suffices to add a pointer to the new policy function to the available strategies list. This function should return a pointer to the chosen scheduler run-queue based on two parameters, the newly spawned actor and its parent.

The application developer can therefore choose, in runtime, the desired initial placement policy. For that we modified the already available `erlang:system_flag/2` function to accept two additional flags. One flag to define the initial placement policy for hubs (`scheduler_ip_strategy_hub`) and other for regular actors (`scheduler_ip_strategy_regular`). In both cases, the available strategy atoms are `default`, `compact`, `scatter`, `circular`, and `random`.

As an example, Listing 5.2 shows how the `default` and `random` strategies were implemented. Lines 1-4 show Erlang's default strategy while Lines 6-9 show the random strategy. The default strategy simply returns the run-queue of the parent actor. For that it employs the auxiliary function `erts_get_runq_proc` (Line 3) that returns the run-queue currently associated to an actor. The random strategy checks and if necessary initializes the random number generator (Line 7) and returns a random run-queue using the auxiliary `ERTS_RUNQ_IX` macro which returns a pointer to the i-th scheduler run-queue (Line 8).

```
1  ErtsRunQueue* proc_sched_ip_default
2    (Process* process, Process* parent) {
3    return erts_get_runq_proc(parent);
4  }
5
6  ErtsRunQueue* proc_sched_ip_random
7      rng_check_initialize();
8      return ERTS_RUNQ_IX(rng_next(erts_no_run_queues));
9  }
```

Listing 5.2: Implementation of the default and random initial placement strategies.

Similarly to the migration strategies, the currently selected initial placement policy can be queried using `erlang:system_info/1` function call, with the atoms `scheduler_ip_strategy_hub` and `scheduler_ip_strategy_regular`.

5.1.1.4 Actor Heap Allocation and Initialization

The actor's heap allocation and initialization is a fundamental part of an actor's creation. If the initial placement strategy decides to place a newly created actor on a NUMA node other than the node in which the scheduler that created it is running, it will also need to allocate the heap remotely. Erlang's framework for memory allocation is quite intricate. It possesses several distinct memory allocators, each one suitable for different memory uses [37], such as a heap allocator, an allocator for binary type data, and an allocator for short-lived data. In this text we are interested in the heap allocator.

The default behavior of the Erlang VM is the creation of an individual allocator private to each scheduler. This is done to improve performance by avoiding contention to the allocator internal data structures. Since allocators are private to each scheduler and each scheduler is bound to a PU, specific scheduler allocations of memory can be made local through the use of Linux's NUMA policy library. During the RE initialization we use this library to enforce local memory allocations for each one of the threads linked to these schedulers. We modified the Erlang VM to accept additional command line options (described at the end of this section) to enable *local memory allocation policy*.

Remote allocations, on the other hand, are more complex. Direct remote memory allocation using OS interfaces (through `mmap` and `mbind` functions) are not as fast as those provided by memory allocators [38]. The problem is not the remote allocation itself as it is the direct use of the `mmap` function. Erlang's heap allocator as well as typical memory allocation libraries, such as glibc's `malloc` [39] and Tcmalloc [40], employ caches of pre-allocated unused memory (using `mmap` or `sbrk`) to mask this slowness. When these libraries receive a memory allocation request, if it can be fulfilled using this cache, no OS call is performed and good performance can be kept. Additionally, when memory is freed it is not immediately released to the OS, it might be put back on the memory allocator cache. Typically these allocators can be parameterized, for example, to define the desirable cache size and policies to deal with cache fragmentation. While several memory allocation libraries optimized for multi-threaded programs exist [BMBW00, Eva06, 40], no widespread NUMA-aware allocator exists. Kaminski [Kam09] presented a study on a NUMA-aware memory allocator, but memory management interfaces such as MaMI [41] and MAi [RMC+09] seem to be the preferred tools currently in use for this job.

One possible solution to solve remote memory allocation performance would be the creation of node-specific allocators. In this setting, each scheduler (or group of schedulers) would have a set of allocators, one for each distinct NUMA node. To avoid contention, these allocators would need to be replicated throughout the schedulers, be it one per scheduler or one per a limited number of schedulers. The problem with this method is that not only does this bring an increased complexity to the RE but also that such a solution would waste memory since for each scheduler (or group of schedulers) caches of remote unused memory would need to be kept. Therefore, as the number of NUMA nodes on the machine increases, so does the waste of memory, rendering this solution not scalable.

Considering that a memory copy operation from the spawning node to the remote node must be invariably done, we have chosen a simpler solution. In our implementation whenever an actor that needs to be placed on a remote NUMA node is spawned, we actually do not initialize its heap. In fact, we keep the data needed to initialize the heap of the newly spawned actor on the spawning node and insert a dummy actor into the target remote run-queue. When this dummy actor is scheduled for execution for the first time, the RE realizes that it is a dummy actor and, using information contained inside the dummy data structure, allocates a new actor and initializes its heap thus finally performing the remote memory copy. Since this part of the process is done on the remote node, all memory allocations are done using the local allocator and the heap and actor itself are automatically allocated on the correct NUMA node. We call this feature *deferred allocation*.

Both local memory allocation policy and deferred allocation only make sense if the schedulers are bound to the PUs of the hardware platform. Thus, to ensure they are only enabled when schedulers are bound we have modified the currently existing scheduler binding command line option `+sbt` to accept some additional choices. The valid prefixes continue to be the already existing binding strategies (*e.g.*, spread→`s`, no-spread→`ns`)[3]. However, the suffixes now indicate which options should be enabled, `df` for deferred allocation and `pp` for local memory allocation policy. For example, the following are all valid initialization lines:

[3] http://www.erlang.org/doc/man/erl.html#+sbt

- `erl +sbt spp`
  Spreads schedulers using local memory allocation policy.
- `erl +sbt nsdf`
  No spread binding type using deferred allocation.
- `erl +sbt tnnpsdfpp`
  Thread no node processor spread binding type using deferred allocation and local memory allocation policy

5.1.2 Experimental Platform

In order to evaluate the proposed modifications, we have used a 4-node NUMA machine with 32 PUs (NUMA 32). Table 5.1 summarizes its general characteristics and Figure 5.1 brings its simplified architectural view. Although this platform possesses hyper-threading capable processors, all of our tests were run with hyper-threading disabled to avoid performance measurement interferences.

The NUMA 32 platform is composed of four nodes each with an eight-core Intel Xeon Beckton X7560 processor. The LLC of each processor is shared by all its cores and the NUMA interconnection is a complete graph.

5.2 Experimental Results

The original 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 tested application execution times without the use of any optional VM parameters. We will refer to this configuration as *default*. However, for the sake of a fair comparison, we present, next to our results, the performance obtained by the best tunning of policies using only the options present on the the original VM. We will refer to this configuration simply as *best tuning*. Similarly, the performance results that made use of the policies we have proposed will be referred to as *proposed*. Table 5.2 describes the policy options we varied during our tests according to their availability on the original and modified versions of the VM. As Cost function, we used the bandwidth between the nodes obtained using HieSchella [36] (*cf.* Chapter 4).

NUMA 32	
NUMA Nodes	4
Cores	32
Frequency	2.27 GHz
Total RAM	64 GiB
L3 Cache	24 MiB
NUMA Factor	1.2 to 3.6
Linux Kernel	3.5.7
GCC	4.7.2

Table 5.1: NUMA 32 platform specifications

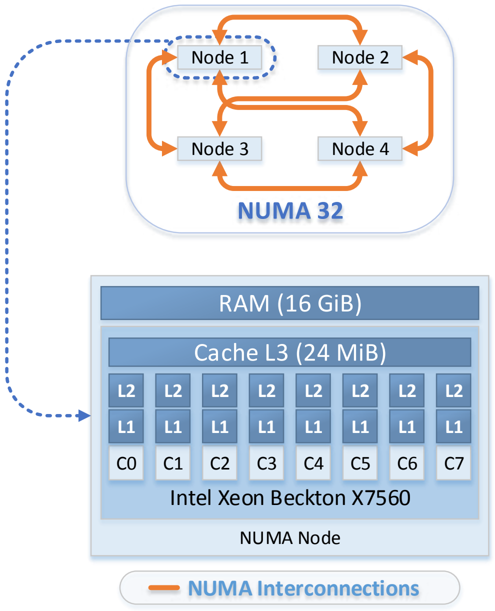


Figure 5.1: Simplified architectural view of the NUMA 32 experimental platform

5.2.1 Benchmarks

To evaluate the performance of our prototype, we have used the BenchErl benchmark suite [APR+12]. 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 actors 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 actors that send at the same time a number of messages to only one receiver. This receiver is clearly a communication hub and its affinity set is the whole set of actors.

	Availability		
Policy	**Orig.**	**Mod.**	**Description**
Scheduler Binding	•	•	Determines how each scheduler should be bound to each PU. By default schedulers are unbound.
Compaction of Load	•	•	Defines if schedulers should be suspended to minimize how often they run out of work. Enabled by default.
Initial Placement		•	Chooses where a newly spawned actor should be placed for its initial execution. Available choices are *default*, *compact*, *scatter*, *circular*, and *random*. It is possible to use this policy for every actor or just for the hubs. In this case, the default policy is used for the regular actors. Requires *Scheduler Binding* option.
Local Memory Allocation		•	Forces schedulers to allocate memory on the same NUMA node to which they are bound. Requires *Scheduler Binding* option.
Deferred Allocation		•	Allows local memory allocation even if an actor was spawned on a node different of the node in which it was first scheduled. Requires *Initial Placement* (different of *default*) and *Local Memory Allocation* options.
Hierarchical Work-Stealing		•	Switches between the default and the hierarchical work-stealing algorithms. Requires *Scheduler Binding* option.
Hierarchical Load-Balancing		•	Switches between the default and the hierarchical load-balancing algorithms. Requires *Scheduler Binding* option.

Table 5.2: Tunning Parameters

► **big** This benchmark creates a number of actors that send, all at the same time, messages to every other actor 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 actor is composed of every other actor in execution.

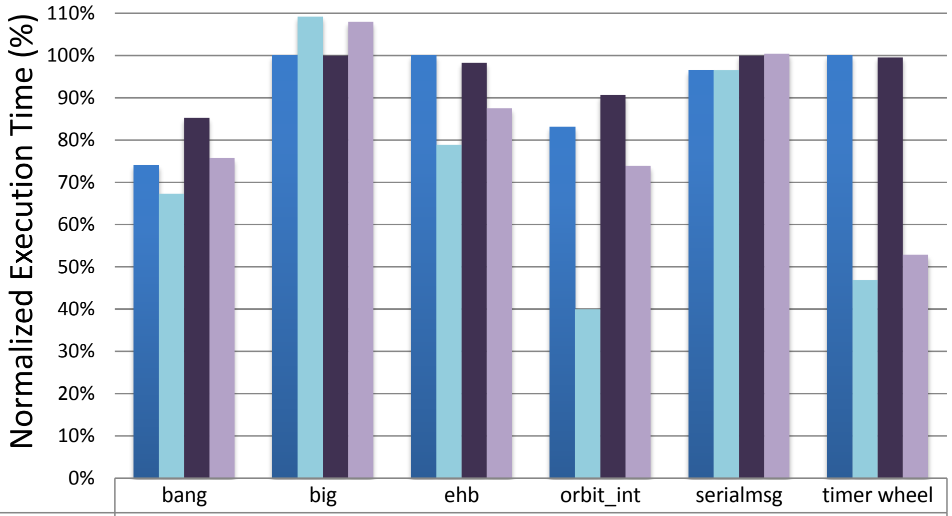
- **orbit_int** It is an implementation of a DHT. In this DHT, each bucket is an actor. To add something to the DHT, an actor sends a message with the data to be inserted to the appropriate bucket. Upon the reception of this message, the bucket/actor might need to process it before storing it. This can take some time thus it is done in parallel by the creation of multiple worker actors. When it is completed, additional data that must be stored in the DHT might have been generated. Their generators, the worker actors, 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 actors are involved in most communications and perform the role of a master actor that spawns several workers.

- **ehb** It is an Erlang implementation of the Hackbench [27] stress test for schedulers. It works creating several groups of communicating actors. Each of these groups has one coordinator that spawns a set of sender and receiver actors (its affinity group) and receives a message back from these actors 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 actor acts as a message dispatcher for every other actor on the system. The benchmark works by creating two sets of actors, the *senders* and the *receivers*. The communication between these two sets of actors is done, exclusively, through this dispatcher. The affinity group of this actor is the whole set of actors.

- **timer_wheel** This benchmark spawns a set of actors that exchange ping and pong messages. Both the ping and pong messages are sent and received by every actor. 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, there is no clear communication hub, and the affinity group of each actor is the set of the remaining actors.

We have extensively tested the modified VM on the NUMA 32 platform. We also show our preliminary results for the Altix UV 2000 platform on Appendix C, a machine with a higher NUMA node count in which neither the original nor the modified VMs were able to efficiently execute. Figure 5.2 depicts the normalized execution time of the chosen benchmarks, with respect to the default configuration (*i.e.*, we take as a baseline the performance of the VM with no optional parameters). We show results for both the short and intermediate data input sizes. All the results presented in this section were obtained from averages of at least 30 samples and analyzed using 95 % confidence intervals to ensure statistical significance. Since there is not much variation in the data, the confidence intervals are too small to be graphically represented. For this reason they were omitted from the figures. For instance, the 95 % confidence interval of the mean execution time for `ehb` with the proposed configuration and the short input size is 0.663±0.002 53 s, which would represent a normalized variation of less than 0.33 % in Figure 5.2.



	bang	big	ehb	orbit_int	serialmsg	timer wheel
🟦 Short - Best Tuning	74%	100%	100%	83%	97%	100%
🟦 Short - Proposed	67%	109%	79%	40%	97%	47%
■ Intermediate - Best Tuning	85%	100%	98%	91%	100%	100%
🟪 Intermediate - Proposed	76%	108%	88%	74%	100%	53%

Figure 5.2: Normalized execution time of the benchmarks for two different data input sizes on the NUMA 32 platform.

In this machine, we were able to to improve the performance in benchmarks: `bang` , `ehb`, `orbit_int`, and `timer_wheel`. Table 5.3 shows the full list of the

	Default		**Best Tuning**	
	short	**intermediate**	**short**	**intermediate**
`bang`	1.48	1.32	1.10	1.13
`big`	0.92	0.93	0.92	0.93
`ehb`	1.27	1.14	1.27	1.12
`orbit_int`	2.50	1.35	2.08	1.23
`serialmsg`	1.04	1.00	1.00	1.00
`timer_wheel`	2.13	1.89	2.13	1.88

Table 5.3: NUMA 32 speedups for the proposed configuration

speedups obtained by the proposed configuration in relation to the default and best tuning configurations.

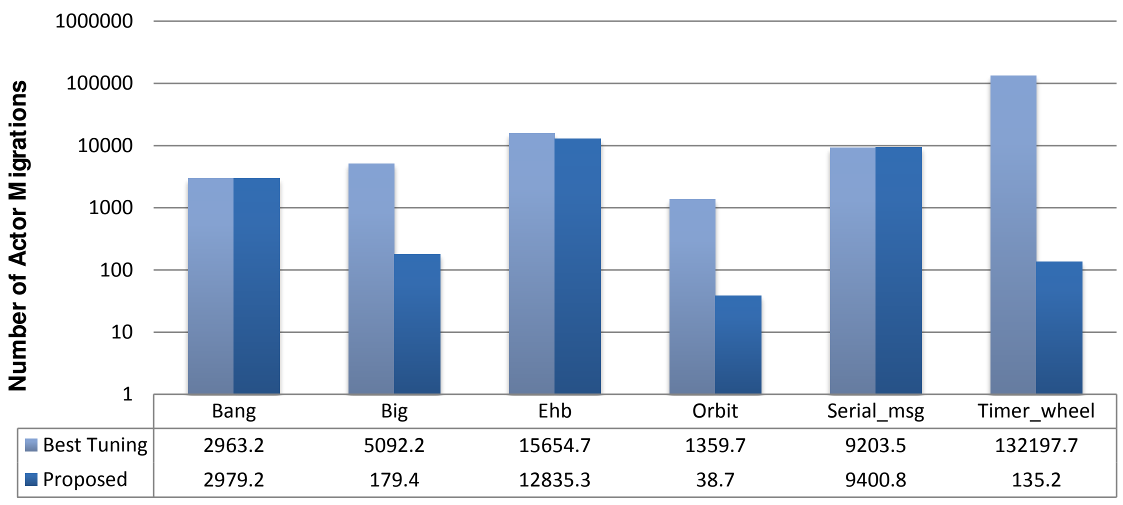
Let us start by analyzing the performance of the `orbit_int` and `ehb` benchmarks. These benchmarks have clearly defined communication hubs: the buckets and the coordinators. These actors are involved in most communications and perform the role of a master actor. Both these benchmarks reflect the fundamental ideas behind our proposal and the considerable speedup we have obtained with these applications shows that when our assumptions fit the target application behavior, our approach has an important impact in performance.

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 actor is composed of every other actor 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 actor of a system is a hub, where no actor 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 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 actors, our approach ends up introducing overheads that we are not be 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 actors. The considerable decrease on the number of actor migrations (see Figure 5.3) is responsible for the reduction of 53 % (short) 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 actors. A bad initial actor distribution means that some schedulers will become overloaded, and thus actors on these schedulers will have a higher level of processor sharing. Since many actors 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 actor distribution means a reduced number of migrations due to load-balancing. An actor migration, by itself, has a cost that is non negligible. Moreover, the execution of the actor on a NUMA node different of his home node imposes additional overheads. Figure 5.3 shows the average number of migrations using the *best tuning* and the *proposed* 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. A curious 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 actors than the available number of schedulers. This renders the compaction of load essentially inactive for most of the time.



	Bang	Big	Ehb	Orbit	Serial_msg	Timer_wheel
Best Tuning	2963.2	5092.2	15654.7	1359.7	9203.5	132197.7
Proposed	2979.2	179.4	12835.3	38.7	9400.8	135.2

Figure 5.3: Average number of actor migrations per execution using the intermediate data set with the *best tuning* and *proposed* configurations of the VM.

5.2.2 Real Applications

Having evaluated our prototype with benchmarks from the BenchErl suite, we proceeded to the evaluation of real applications. For this evaluation we used the two applications already presented in Chapter 3: Sim-Diasca and ErlangTW. We now further detail the characteristics of these applications.

► **Sim-Diasca** This application is a DES engine that can perform simulations on local and distributed environments [26]. In this document we limit ourselves to the local case. Its major use has been to evaluate the performance of communication networks to be used by future large scale energy distribution smartgrids [SKZ+11]. Unfortunately, access to the data needed to perform these specific simulations is restricted. We therefore evaluated Sim-Diasca using another simulation case, the City Waste Management Simulation [Bou13]. This simulation creates a number of waste sources and the entities that deal with the waste such as incinerators, landfills, waste trucks, and roads. Each one of these entities has specific characteristics that must be taken into account by the simulator. For example, waste sources might create varying amounts of waste depending if they are residential or industrial, trucks can transport waste between points of interest using roads. The average speeds of the trucks are modeled as a function of the road capacity and its utilization. Trucks play a central role in this simulation. These entities exchange messages with almost every other entity types such as waste sources, incinerators, landfills, and roads. When we analyze the communication graph of this application, trucks stand out as hubs (*cf.* Figure 3.6(b)). We slightly modified Sim-Diasca's code to include the necessary hints to mark trucks as hubs. For our measurements we used Sim-Diasca version 2.2.0 with the default simulation datasets.

- **ErlangTW** Similar to Sim-Diasca, this application is also a parallel DES simulator engine [TDM12]. On the other hand it performs simulations using a Time Warp synchronization protocol rather than the time-stepped approach used by Sim-Diasca. This DES works by partitioning the simulation model into submodels, called Logical Processes (LPs). Since the processing of each one of these submodels can be done in parallel, in ErlangTW LPs are represented by actors. During the execution, each LP becomes responsible for a set of entities. Using the LP, these entities exchange timestamped messages representing the simulated events. The LPs are central pieces of the ErlangTW software architecture, and are involved in most communications. We therefore introduced the appropriate calls into ErlangTW's code base to mark LPs as hubs. We have exercised this simulator using the default PHOLD Benchmark [Jon86] that comes bundled with ErlangTW software package. Our tests employed ErlangTW version 1.0. Although our experimental hardware platform features 32 PUs, this version of the simulator is limited to the creation of only 25 LPs. Any number higher than 25 LPs makes the simulation fails.

Figure 5.4 shows the normalized execution time of the applications with respect to the default configuration, *i.e.*, taking as the baseline the execution time using the VM with no optional parameters. For Sim-Diasca we show improvements of ∼8 % and ∼5 % in execution times when we compare to the default and best-tunning configurations. While these performance improvements are not as prominent as those we obtained with the BenchErl benchmarks, they are still significant if we consider that they were obtained with just minor modifications to the simulation code, *i.e.*, the inclusion of the hints to the actor RE. For Sim-Diasca the factors

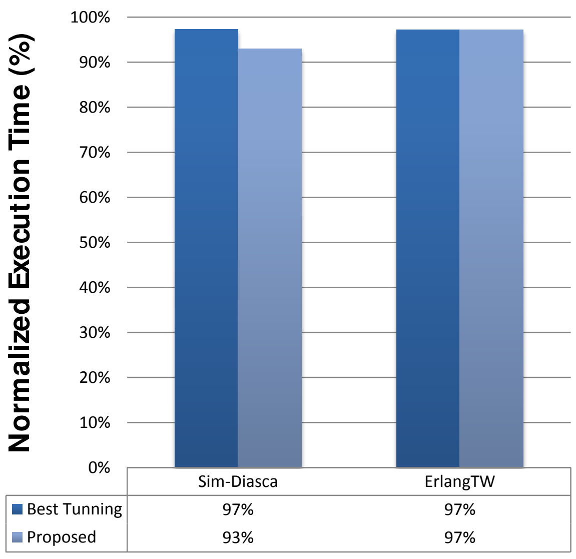


Figure 5.4: Normalized execution time of Sim-Diasca and ErlangTW on the NUMA 32 platform.

	Default	**Best Tunning**
Sim-Diasca	1.08	1.05
ErlangTW	1.03	1.00

Table 5.4: Sim-Diasca and ErlangTW speedups for the proposed configuration on the NUMA 32 platform.

that influenced performance the most where, in order, hierarchical work-stealing, hierarchical load-balancer and scheduler binding.

As for ErlangTW, we were able to improve the performance in only ∼3 %, approximately the same improvement we obtained using only the options available on the original VM. Table 5.4 brings the full list of speedups for these two applications considering the default and best-tunning configurations as baselines.

While investigating the reasons why the performance was practically unchanged in ErlangTW, we realized that not only it currently has a limit on the number of LPs, but also that it uses a non-traditional message-processing approach. The relevant excerpt from the ErlangTW source code is shown in Listing 5.3.

Tail-recursive function calls with receive blocks are the traditional implementation technique used in Erlang to create an actor event-loop. The `receive...after` construction is also quite common. In this case, an actor that has no message to process can continue to do something else (after a timeout) and check for messages at some point later in time. A timeout 0 (Line 8) means that it does not wait if there are no messages in the mailbox. However, the way ErlangTW employs it, makes the event-loop of actor to be always active, even if it has nothing to process, since

```
main_loop(Lp) ->
    receive
        Message ->
            NewLp = Lp#lp_status{
                received_messages=queue:in(
                    Message, Lp#lp_status.received_messages)},
            main_loop(NewLp)
    after 0 ->
        main_loop(process_top_message(
            process_received_messages(
                Lp, Lp#lp_status.max_received_messages)))
    end.
```

Listing 5.3: ErlangTW excerpt from the LP main loop

it never pauses to wait for messages. It is always either checking for new messages or trying to process a, potentially empty, buffer of received messages. We tried to contact ErlangTW developers to understand their motivations for such a construction, but up until now we had no response. One of the possible reasons the developers could have had to code the application this way was to try to avoid the overhead of being scheduled out when waiting for a message. Whenever an actor issues a receive operation on the Erlang VM, and there is no message in the mailbox that satisfies this request, the actor is scheduled out until a new message is received. When the message is delivered, the actor is put back into the run-queue and to execute it has to wait until it gets scheduled again.

Unfortunately, an active-loop wait may cause severe performance side-effects in the RE. For example, if more LPs than the number of available schedulers are created, each actor will consume its full reduction quota (even if it has no message to process) before giving up the processor to the next actor. If the next actor also has no messages to processes, this process is once again repeated. To show how severe the performance degradations can be, we executed the same simulation using 25LPs on 25 schedulers and afterwards using only one scheduler. We expected the execution with one scheduler to take approximately 25× more time. That was not, however, the observed result. With 25 schedulers the simulation finished in ∼14 s while the same execution on one scheduler took ∼2759 s, *i.e.*, 7.9 times more than the expected execution time.

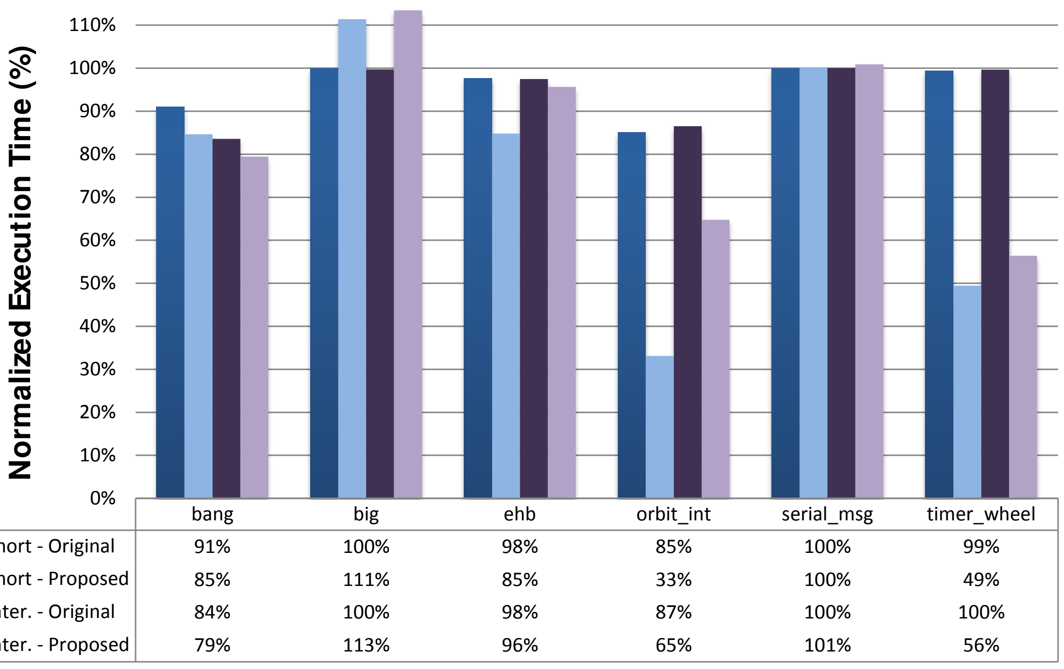
The results obtained by the execution of Sim-Diasca and ErlangTW shows us that, when the application fits our assumptions about the problem domain, we can indeed create an actor RE with better performances on NUMA platforms.

5.2.3 Original VM Comparison

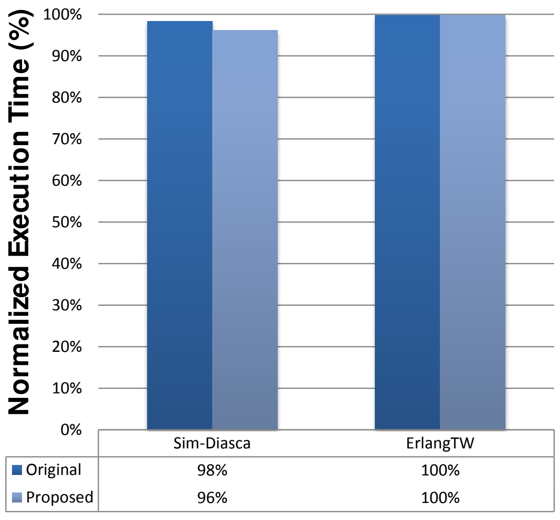
The performance results we presented, although measured using the default behavior of the Erlang VM, were assessed using the modified VM. This was done not only due to the limitations of the unmodified VM (for example, there is no support to trace actor migrations), but also because we are interested 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 original 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 % depending on the application behavior. 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 5.5 depicts the execution of the benchmarks (using two distinct workloads) and the real applications. We show in each case the performance with the best configuration we could find for the original VM as well as the best configuration found using the modified VM, including the parameters dealing with the proposed modifications. Execution times were normalized by the original VM execution time with the default configuration, *i.e.*, with no optional parameters.

5.3 Concluding Remarks

In this chapter we presented our prototype for a NUMA-aware actor RE. We modified the Erlang VM to implement the proposed optimizations and evaluated the prototype using standard benchmarks. Additionally, we have made it simple to add new strategies for work-stealing, load-balancing and initial placement of actors. Among other modifications, we can highlight the addition of hierarchical work-stealing and load-balancing, scatter and compact strategies for the initial placement of actors, deferred allocation of heaps and local allocation of memory on NUMA platforms.



	bang	big	ehb	orbit_int	serial_msg	timer_wheel
Short - Original	91%	100%	98%	85%	100%	99%
Short - Proposed	85%	111%	85%	33%	100%	49%
Inter. - Original	84%	100%	98%	87%	100%	100%
Inter. - Proposed	79%	113%	96%	65%	101%	56%



	Sim-Diasca	ErlangTW
Original	98%	100%
Proposed	96%	100%

Figure 5.5: Normalized execution time of BenchErl benchmarks, Sim-Diasca, and ErlangTW using the best parameterization for the original and modified VMs on the NUMA 32 platform. Execution times were normalized by the original VM execution time with no optional parameters.

To assess the impact of our optimizations, we evaluated the performance of the modified Erlang VM using standard benchmarks and two real applications. Our experiments show that, if the target application fits our assumptions about the problem domain, NUMA-aware optimization policies can bring significant performance improvements to the RE and therefore to the applications.

Published research on the efficient implementation of actor REs on NUMA platforms is quite scarce. On the other hand, research on efficient utilization of hierarchical machines in other contexts is quite common. In the next chapter we outline and compare these research efforts to our own.

CHAPTER 6

Related Work

RESEARCH on the efficient use of hierarchical shared memory platforms is a relatively popular domain of research. This interest can be explained by their ever-increasing adoption and by their distinctive memory access characteristics. Despite this popularity, published research on the efficient use of NUMA platforms by actor runtime environments is limited. In this chapter we describe some of the most relevant related works, specially those that deal with the runtime aspects explored in this thesis, *i.e.*, memory access and communication optimization.

We divide the description of these works in two sections. The first, with research directly related to actor REs and the second where we list general NUMA-aware performance improvements approaches.

6.1 Actor Related Approaches

A fair number of actor-based concurrency frameworks exist, be them actor-based languages or actor based frameworks for general programming languages. It would be impossible to discuss them all in this section. For this reason, we discuss a few selected works for which we highlight some of the similarities and differences to our own approach. Although highly parallel actor REs have frequently been the subject of research, most of these works deal with the efficient use of multi-core machines and the development, analysis, and optimization of these REs for SMP platforms.

6.1.1 Erlang VM

Several works on the performance improvement of the Erlang VM for SMP platforms exist. Originally, the Erlang VM allowed the creation of many actors, however their execution was sequential as their time-shares to use the PU were interleaved using a single OS thread. Among the earliest works, Hedqvist [Hed98] presents an initial parallel event-based single-queued implementation for the Erlang VM. Improving on this work, Lundin [Lun08] describes the passage from the use of single-queue to multiple-queues to remove performance bottlenecks created by accesses to this single data-structure.

Using some of the benchmarks we have also employed in our evaluation, Zhang [Zha11] characterizes the scalability of the Erlang VM on the TilePro64 many-core processor. In this work, he concludes that the overhead imposed by uncontended locks is a major source of performance loss in this system. However, he limits himself to the analysis of the performance of the VM without providing any platform-specific optimization such as a better use of the processor Network-on-Chip (NoC). As an attempt to decrease the contention caused by locks in the context of ETS tables[1], Nyblom [Nyb11] proposes the use of software transactional memory to control accesses to the common data structure with some encouraging results.

Performance and memory consumption trade-offs between private, shared and hybrid heap architectures in concurrent message-passing languages are explored by Carlsson *et al.* [CSW03]. Johansson *et al.* [JSW02] evaluate the performance of these heap architectures on the Erlang VM. In order to do this evaluation, the authors employ synthetic benchmarks as well as real applications. Their final conclusion is that best choice ultimately depends on the characteristics of the application, however, if the choice has to be made beforehand, the shared heap architecture seems to be the best as it usually offers better memory utilization and performance. These tests were performed using a regular SMP machine, therefore the results do not take into consideration the NUMA effects on the application execution.

The Release Project [BCC$^+$12, 42] 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

[1] http://www.erlang.org/doc/man/ets.html

suite of benchmarks [APR+12] we used in our tests has been created by this project. Several important performance aspects related to the Erlang VM were tackled. As some examples we can cite native code compilation, message exchange optimizations, profiling and distributed execution [PRL13, SST12, LT13, PS12]. Although all of these aspects can be used to improve the performance of the Erlang VM running on NUMA platforms, they are also generic in the sense that they are not NUMA specific. They are as important on NUMA platforms as they are on SMP platforms.

6.1.2 Other Runtime Environments

In this section we present some REs that have similar characteristics to our work. We start by the introduction of two actor REs, Kilim and Akka, and conclude with the discussion on Charm++. Although Charm++ is not, strictly speaking, based on the actor model, it still shares many of its characteristics and possesses a few approaches to NUMA platforms similar to our own.

► **Kilim** [SM08] is an actor development framework for Java. It provides lightweight event-based actors, zero-copying message exchange, and isolation-aware messaging. In order to provide message exchanges between actors and immutability guarantees, it imposes some restrictions on pointer aliasing inside messages. Its technique is, in fact, very similar to the approach used by hybrid-heap based message exchanges presented in Chapter 4. Kilim terminology is a little different from the terminology used in this thesis. In this system, a *scheduler* is a bundle composed of a thread-pool, a scheduling policy and a collection of runnable actors.

Threads might be bound to the PUs and, by default, the RE creates as many threads as the number of PUs in the system. However, in case of need, the pool might be grown. Actors in the runnable actors collection are cooperatively-scheduled according to the scheduling policy. By default, actors in this collection are scheduled in a round-robin fashion although other policies are also available and can be defined by the application itself.

One could, therefore, classify Kilim's default behavior as an event-based single-queued actor RE. On the other hand, the RE allows the creation of new schedulers in runtime with an arbitrary thread-pool size. In essence, this makes Kilim capable of running as a multiple-queued RE. However, to use these application-created schedulers, a newly spawned actor has to explicitly specify the target scheduler. Although manual, actor migrations (or *scheduler hopping* in Kilim's parlance) between schedulers are possible. Scheduler hoping is indeed touted as an efficient way to perform data partitioning, that is, to place actors near the data and therefore provide better cache locality [Sri10].

► **Akka** [Gup12] has a concept similar to Kilim's schedulers called *dispatcher*. Dispatchers are responsible for actor scheduling and the distribution of the messages that need to be processed. By default Akka provides four kinds of dispatchers: default, pinned, balancing and calling thread. The default dispatcher is an event-based single-queue implementation backed by a thread-pool of configurable size. When an actor is created and no dispatcher is set, this is, as its name states, the default behavior. The pinned dispatcher is a thread-based implementation, therefore each actor has its own OS thread. The balancing dispatcher employs an event-based approach, however, it has a peculiar behavior. This dispatcher creates only one mailbox that is shared by every actor associated to it. When a message is received, an idle arbitrary actor will be selected to process it. Thus, it is expected that every actor on the same dispatcher be capable of processing every received message. Finally, the calling thread dispatcher, as the name states, has no OS thread associated to it. This dispatcher executes the message processing using the thread of the message sender and is mostly used for debugging purposes [13].

Both Kilim and Akka have application level mechanisms that allow developers to bind the execution of a group of actors to a specific NUMA node. This allows application developers to manually improve overall system performance in these platforms. A possible use of these mechanisms would be the manual creation of a scheduler (Kilim)/dispatcher (Akka) for each NUMA node. That way, actor affinity could be easily maintained. On the other hand, load balancing would be need to be manually kept by the application developer. A simply manual intervention might work for a specific application and hardware platform, but such a solution lacks portability. A more complex manual solution is, in our opinion, undesirable since we want the application developer to focus on the application and not on the execution environment in which this application will be run.

► **Charm++** is a parallel programing language based on C++ [KK93]. Applications running on this platform are composed of several communicating entities called *chares*. Chares communicate through asynchronous message-passing, they can be migrated and, as in the actor model, the RE ensures location transparency. On a multi-core shared memory machine, the RE creates as many threads as PUs. Each thread has an individual queue were ready-to-run chares are kept. There is no preemption and, therefore, chares are scheduled cooperatively either explicitly yielding control to the RE or being interrupted when some specific RE calls are made.

Charm++ has a concept similar to the initial placement policy we defined. In Charm++ this concept is called *seed load-balancers* [43]. Charm++ comes with a few pre-defined seed load-balancers such as *random*, *neighbor*, *spray* and *workstealing*. As the name states, *random* chooses a random processor. *Neighbor* takes into consideration the virtual topology of the processors and, when overloaded, processors migrate chares exclusively to their direct neighbors. *Spray* takes into consideration the average load of each processor to determine how chares should be distributed. Finally, as the name states, in the *workstealing* strategy idle processors steal chares from loaded processors.

Periodical load-balancers are also present in Charm++. Pilla *et al.* [PRC+12] proposed an alternative load-balancer which takes into consideration the asymmetrical NUMA communication costs, application communication graphs and migration costs to define migrations. Similar to the actor applications we analyzed, the number of chares in a Charm++ application is typically big enough to make exact optimization solutions impractical. Their solution is therefore based on a simple heuristic that iteratively maps chares, from the most to the least CPU intensive, to the PUs in a way that minimizes the execution cost of each chare. The execution cost is estimated as a function of the load of the PU, and the amount and cost (based on the topology of the hardware platform) of communications between chares. With this load-balancer the authors were able to achieve speedups up to 20 % when compared to state-of-the-art load-balancers. Additionally, some of the provided hierarchical load-balancers divide processors into hierarchically arranged groups in which load-balancing happens independently. This scheme is similar to the hierarchical load-balancer we proposed in the sense that global load-balancing is only used as a last resort.

Even if chares are very similar to actors, there are still some differences. Since its inception Charm++ supports abstractions for distinct levels and modes of information sharing, *i.e.*, message exchanges are not the only means of communication between chares. Its authors' opinion is that having only message exchanges diminishes both the efficiency as well as the expressiveness of the language [KK93]. This is, in fact, one of the explanations Tasharofi *et al.* [TDJ13] found to the fact that Scala applications frequently mix the actor model with other concurrency models.

Scheduling of chares is message-driven, *i.e.*, there is no preemption and context switching is cooperative. In actor REs preemption is either enforced by the RE or the RE takes proactive action to avoid actor starvation such as the creation of additional threads. Additionally, actors are typically finer grained: an actor can be created to perform a specific very short task. Moreover, while the state of the system is normally kept by actors, this is not the typical case in Charm++ applications.

6.2 General NUMA-Aware Parallel Approaches

Researchers from the parallel processing community have shown a strong interest for non-actor platform-aware parallel solutions for an efficient use of hierarchical shared-memory platforms. These solutions range from high-level RE based solutions to low-level OS related approaches. We will now explore some of these solutions in more detail.

► **MPI** [For12] is among the predominant messaging standards for HPC applications. The MPI standard defines a set of functions to specify the topology of the underlying platform. These functions can be used to create a specific MPI communicator in which the process ranks are reordered for a better process-to-PU mapping. Rashti [M.J11] *et al.* show how a better match between the MPI process ranks and the physical topology can bring considerable gains in communication performance. In this work, however, the authors do not consider the communication graph of the application. In a similar approach, after a preliminary application profiling phase, some works [MCO09, JMT13, LWZ13] perform process rank reordering taking into consideration not only the network and the characteristics of the NUMA platform but also the communication graph of the applications, with good performance results. In these approaches, the authors employ graph partitioning and tree-matching algorithms to find good process placement solutions. Differently from the actor model, MPI processes are created at the beginning of the execution. Process placements are calculated *a priori*, the mapping happens at initialization and processes remain bound until the end. The granularity of a MPI process is much coarser than that of an actor and it is this characteristic that allows the use of optimization techniques, instead of heuristic ones, to obtain results in an reasonable amount of time. Additionally, since processes are bound from the outset, the optimal binding is in fact that with the best average performance throughout the whole execution.

► **Java** garbage collection mechanisms have been used by Ogasawara [Oga09] to create a NUMA-aware memory manager for that language. In this innovative approach, the memory manager performs object-placement decisions during the garbage collection cycles. During the garbage collections, objects are marked according to the threads that have access to them. Among these threads, the one that accesses it the most is defined as the dominant thread of that object. Next, the NUMA-aware garbage collector copies this object from its current location to the NUMA node in which the dominant thread is located. Similar to our NUMA-aware allocators, the author implemented a variant of the memory allocator so that when memory is allocated by a thread it is done preferably from the NUMA node in which the thread is executing.

► **OpenMP** [CCD+06] research on efficient execution on NUMA platforms is abundant. Nikolopoulos *et al.* [NPP+00] present a page migration mechanism based on a initial profiling of the first iterations of an OpenMP application. Using code-instrumentation, it detects which threads access each page of memory and migrates them accordingly. Duran *et al.* [DPA+08] propose a few extensions to the OpenMP standard so that the affinity between threads and data can be traced. Using BubbleSched [TNW07] for hierarchical grouping and placement of threads and MaMI [41] to perform memory migrations, ForestGOMP [BFG+10, BAG+10] presents a multi-level thread scheduler combined with a NUMA-aware memory manager. It uses bubbles to cluster threads and schedules work using a work-stealing algorithm which chooses for migration those threads that have the least amount of memory associated with it. NUMA-aware scheduling of OpenMP tasks is dealt with by Broquedis *et al.* [BFG+09] and by Olivier *et al.* [OPW+12]. However, the task model presented by OpenMP is different from that of the actor model in very important aspects such as granularity and communication.

► **Kernel** level solutions such as AutoNuma and NumaSched [SS12, 44, 45] show that improvements on the REs are not the only possibilities currently in exploration. These approaches try to transparently improve performance by doing better memory allocation and process 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.

Despite the existence of several solutions for the efficient use of NUMA machines, we consider our proposal to employ the actor model in these hierarchical platforms essential. Contrary to most alternatives, the actor model offers a high-level programming interface and therefore allows the software developer to write applications that are totally decoupled from the underlying hardware architecture. Additionally, as the availability of hardware platforms with a high number of cores and processors continues to increase, so will the search for easy-to-use parallel solutions with good performance.

CHAPTER 7

Conclusion and Perspectives

HARDWARE manufacturers compete in an endless race to produce faster processors. In order to overcome the growing design complexity and prohibitive power consumption, one of their latest strategies in this race is the creation of processors with an ever increasing number of cores. This strategy, in effect, shifted the burden of parallelization from hardware designers to software developers.

Parallel programming tools have existed for a long time. In spite of that, concurrent and parallel aspects of software development have for many years been a niche and essentially have been ignored by most software developers [Sut05]. It was not until multi-core processors hit the mainstream market that we begin to see a general increase in the interest for these tools. Lately, the actor model, which had been created in the early seventies [HBS73] (mid-eighties if we consider it in the form we now take for granted [Agh86]), has seen an unprecedented rate of adoption due not only to its intrinsic characteristics (absence of shared state and therefore no locks or synchronization) but also to the emergence of high performance actor REs such as the Erlang VM.

As the adoption of the actor model grew, actor applications gradually began to make their appearance on platforms with higher processing capacities. These platforms not rarely are hierarchical shared memory multi-core, or simply NUMA, machines. According to what was presented in this text, while actor REs optimized for SMP architectures are quite common, the same cannot be said about NUMA-aware REs. This is true despite the fact that several mission critical actor applications run on NUMA platforms.

In this thesis we analyze some of these actor applications searching for common characteristics. With the knowledge about applications behavior and about underlying hardware platform, we propose, realize and evaluate a set of improvements using a real actor RE, the Erlang VM. These improvements take into consideration application and hardware idiosyncrasies to make better scheduling, load-balancing and memory management decisions. These modifications were done on the RE level, therefore no alterations on the applications themselves are needed. However, the application developer can also provide the RE hints about the behavior of his application, helping the RE take better decisions and therefore improving the results of our approach.

7.1 Contributions

After a brief introduction (Chapter 1) and some background (Chapter 2), we present the first contribution of this thesis. The analysis and characterization of actor applications (Chapter 3). We first analyze general actor characteristics such as lifespan, message sizes and communication costs. Then, we stand back to observe more general execution properties of actor applications, specially actor interactions. During this observation we realize that some actors are much more connected than others. These actors are responsible for the creation of the majority of other actors and are involved in most communications. We call these special actors, hubs. Our analysis also shows us that actors tend to communicate only within a small subset of other actors. This small subset is typically created by a hub. We define the set of actors with which an actor communicates as the actor's affinity group.

In, Chapter 4 we take the findings of our previous contribution and the knowledge of NUMA architectures into consideration to propose a concrete hierarchical approach to the creation of an efficient actor RE. We discuss initial placement policies and the concept of scheduler distances, giving the application developer the liberty to choose whatever distance function is more appropriate to his context. The values provided by this function are central to our hierarchical approach to actor migrations. Then we present our hierarchical load-balancer and work-stealing algorithms. These migration algorithms try to keep actors near the node in which their memory is allocated, *i.e.* its home node. To that aim, the proposed migration algorithms not only avoid to migrate actors away from their home nodes but also try to bring them back. When migration between NUMA nodes is unavoidable, the algorithms try to minimize the distance between the actor and its home node. For that they make use of the pre-calculated distance vector of each scheduler.

We applied and evaluated the performance of our proposal for a NUMA-aware actor RE to the Erlang VM. To assess the impact of our optimizations, we evaluated the performance of the modified VM using standard benchmarks and real applications. Our experiments successfully show that, when we consider the default Erlang VM as the baseline, some simple NUMA-aware optimization policies can bring significant performance improvements (up to a factor of 2.50). We have also shown that poorly coded applications, that might display acceptable performances on SMP architectures, might experience severe performance losses on these hierarchical shared memory platforms.

In addition to the new strategies for work-stealing, load-balancing and initial placement of actors, we have added new features to the Erlang VM such as local memory allocation and deferred heap allocation. For that we employed the Strategy design pattern, meaning that new policies can be easily created. In order to implement our modifications, we also developed a few satellite tools for tracing and visualization of communication graphs (used to generate Figure 3.5) and actor pinning. Some other tools include a simple hardware performance counter tool (used to generate Figure 3.4), and Jhwloc a simple Java Interface for hwloc.

Most of the contributions above were described and published at international events. Appendix A brings the full list of papers published during the elaboration of this thesis.

7.2 Future Work

Even if the approach we proposed is already enough to bring significant performance enhancements to actor REs, it also leads us to a number of research perspectives. One of these perspectives is to make all the new features we introduced part of the official Erlang OTP VM. Currently, the modified VM is available as 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. These changes include making the code compatible with other operating systems such as FreeBSD and improving the support for different hardware architectures. We also envision the integration of our approach into other actor runtime environments such as Kilim and Akka. Additionally, a deeper performance comparison of high-level REs such as those supporting Erlang and Akka to more lightweight approaches such as Charm++ could shed some light on the overhead incurred by the choice of a higher level environment.

In this work we tackled several execution aspects of an actor application. In particular, we analyzed and proposed a novel approach to deal with hubs and affinity groups, we discussed actor communication performance and showed how these aspects could be taken into consideration to decrease the overhead imposed by hierarchical shared-memory architectures. We now present some possible research perspectives on these topics.

► **Hubs and Affinity Groups -** Identification of hubs on our current approach depends on hints provided by the application developer. In this work we analyzed the performance gains a correct and lightweight identification can have on an actor application. A dynamic hub detection and placement mechanism would make our approach even more transparent to the user. There are, however, many challenges to the automatic detection of hubs. These difficulties are mainly due to the size and to the dynamic nature of actor communication graphs. Furthermore, after hubs are initially placed for execution, no re-balance of hubs is performed. Therefore, currently, hubs have their execution restricted to the scheduler chosen by the initial placement policy.

We assume that hub actors are, for most of the time, responsible for the creation of the majority of the actors that belong to its affinity group. This is based on the heuristic design guidelines proposed in Chapter 4. However, this heuristic may not be true for every actor application. 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. For that we could employ graph partitioning techniques on the communication graph using tools such as Scotch [46] in an approach similar to that of Jeannot *et al.* [JMT13] and Li *et al.* [LWZ13].

► **Communication on Hierarchical Platforms -** Affinity groups indicate which actors are more likely to communicate between themselves. Communication performance depends not only on keeping actors near their affinity group but also on the memory organization chosen by the actor RE. Actor heap organization and their impact on the RE performance have been thoroughly studied on SMP platforms [CSW03, JSW02]. However, further research on the impact different heap organizations cause on REs running on NUMA platforms is needed. Even if a global shared heap is not the best option for a NUMA machine, message exchanges could be further improved by the implementation of a shared heap architecture for actors inside the same affinity group, or inside the same NUMA node, in which messages would be passed by value. Additionally, as a runtime optimization, the overhead caused by the use of regular locks to access actor mailboxes could be possibly decreased by the use of software transactional memory [DDS+10, HLR10] or lock-free data structures.

Actor message exchanges are not, however, the only communication costs incurred by these applications on hierarchical platforms. 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. An actor 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 performance presented by the Altix UV 2000 platform (*cf.* Appendix C) is clearly required.

On a many-core processor, communication between cores is accomplished through the use of a NoC. Often the NoC interconnections are not a full graph, therefore communication performance between cores may depend on the topological distance between them. For instance, Tilera's Tile-Gx family of chips employs a grid interconnection between cores [47]. On the other hand, Kalray's MPPA-256 many-core [ABB+13, DML+13] employs a 2D torus topology. NoC interconnections resemble those of a NUMA platform and, for this reason, we intend to further study the adaptation of our approach to many-core platforms such as MPPA-256 [FGM13b].

APPENDIX A

List of Publications

THE following papers were published as a direct result of the research developed during the thesis:

- E. Francesquini and A. Goldman. *Scheduling on multi-core clusters*. Workshop on Algorithms and Techniques for Scheduling on Clusters and Grids, Proceedings of ASTEC. Les Plantiers, France, 2009.
- H. Vincent, V. Issarny, N. Georgantas, E. Francesquini, A. Goldman and F. Kon. *CHOReOS: Scaling Choreographies for the Internet of the Future*. Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing. Bangalore, India, 2010.
- E. Francesquini and A. Goldman. *Análise de Desempenho e Escolha Dinâmica de Escalonamentos para Sistemas Multicore*. II Escola Regional de Alto Desempenho de São Paulo, Anais da II Escola Regional de Alto Desempenho de São Paulo. São José dos Campos, Brazil, 2011. *Best Graduate Student Paper Award*.
- E. Francesquini, A. Goldman and J-F. Méhaut. *Towards Automatic Actor Pinning on Multi-core Architectures*. Proceedings of the 11th ACM SIGPLAN Erlang Workshop, Erlang'12, Copenhagen, Denmark, Sep. 2012.

- E. Francesquini, A. Goldman and J-F. Méhaut. *Actor Scheduling for Multicore Hierarchical Memory Platforms*. Proceedings of the 12th ACM SIGPLAN Erlang Workshop, Erlang'13, Boston, US, Sep. 2013.

- E. Francesquini, A. Goldman and J-F. Méhaut. *A NUMA-Aware Runtime Environment for the Actor Model*. Proceedings of the 42nd International Conference on Parallel Processing, ICPP 2013. Lyon, France, Oct. 2013.

- E. Francesquini, A. Goldman and J-F. Méhaut. *Improving the Performance of Actor Model Runtime Environments on Multicore and Manycore Platforms*. Proceedings of the 3rd International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE) held in the Annual Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH). Indianapolis, US, Oct. 2013.

- M. B. Castro, E. Francesquini, T. M. Nguélé and J-F. Mehaut. *Analysis of Computing and Energy Performance of Multicore, NUMA, and Manycore Platforms for an Irregular Application*. Proceedings of the IA3 Workshop on Irregular Applications: Architectures & Algorithms, held in the Supercomputing Conference (SC'13). Denver, US, Nov. 2013.

- M. B. Castro, E. Francesquini, T. M. Nguélé and J-F. Mehaut. *Multicœurs et Manycœurs: Une Analyse de la Performance et de l'Efficacité Énergétique d'une Application Irrégulière*. Proceedings de la Conférence de Recherche en Informatique, CRI'2013. Yaoundé, Cameroon, Dec. 2013.

APPENDIX B

Erlang's Migration Parameters Calculation

DURING its execution, the Erlang VM gathers and stores statistics about the schedulers and their run-queues. Some of the collected data, described below, are used to calculate the migration limit and migration paths for each one of the run-queues. The collected data is categorized in intervals. An interval is the period of time between each invocation of the load-balancing routine. For this explanation[1] we assume that the number of schedulers n remains constant during the execution, *i.e.*, Erlang's compaction of load algorithm is disabled.

B.1 Migration Limits

Considering $P = \{low, normal, high, max, port\}$ as the set of all possible actor priorities, the collected statistics include for each scheduler $i, 1 \leq i \leq n$:

- $MaxRQSize_{i,p}$ – the maximum run-queue size with priority p reached in the last interval for scheduler i

[1]This explanation is based on the analysis of the Erlang VM codebase, on discussions we had at the Erlang mailing list with Rickard Green (member of Ericsson's Erlang development team), and on the work by Zhang [Zha11].

- *OutOfWork*$_i$ – a flag (0 or 1) indicating if the scheduler i ran out of work in the last interval
- $HRed_{i,p,w}$ – the historic number of executed reductions for the scheduler i, for the run-queue with priority p on the w-th last interval. For example, $HRed_{i,p,0}$ is the number of reductions for the current (last) interval, $HRed_{i,p,1}$ the reduction count for the interval before the last, ...

Using this information, the load-balancer can then calculate a target number of actors per queue. To calculate this number, it uses a few helper functions that we describe now.

The Erlang VM treats regular actors (processes) differently of I/O actors (ports). The load-balancer algorithm differentiates these actors through their priority (I/O actors have the *port* priority). $ActorRed_i$ represents the total number of reductions executed only by regular actors. Execution of I/O actors is considered as the highest priority and the execution of regular and I/O actors are interleaved by the VM.

$$ActorRed_i = \left(\sum_{p \in P} HRed_{i,p,0} \right) - HRed_{i,port,0} \tag{B.1}$$

The availability of a scheduler i for actors with priority p, based on the number of reductions, is defined as follows. First, if in the last interval the scheduler i did not execute any reduction, its availability is 0. For the maximum priority queues (max and port), by construction, it was always available. Therefore the availability is 1. For the remaining priorities, the availability is calculated as a ratio of the total number of regular-actor executed reductions for the run-queues below or equal to that priority level and the total number of regular actor reductions. Equation B.2 formalizes this definition.

$$RedAvail_{i,p} = \begin{cases} 0 & \text{if } \sum_{q \in P} HRed_{i,q,0} = 0 \\ 1 & \text{if } p \in \{max, port\} \\ \dfrac{\left(ActorRed_i - HRed_{i,max,0}\right)}{ActorRed_i} & \text{if } p = high \\ \dfrac{ActorRed_i - HRed_{i,max,0} - HRed_{i,high,0}}{ActorRed_i} & \text{otherwise} \end{cases} \tag{B.2}$$

Additionally, let $AvgHRed_i$ (Equation B.3) represent the historical average number of executed reductions for every priority for the last eight intervals for scheduler i, and *FullSchedulers* be the set of all schedulers that did not run out of work in the last interval (Equation B.4):

$$AvgHRed_i = \frac{\sum_{w=0}^{7} \sum_{p \in P} HRed_{i,p,w}}{8} \tag{B.3}$$

$$FullSchedulers = \left\{ i \,\middle|\, i \in [1, n], OutOfWork_i = 0 \right\} \tag{B.4}$$

Then, we define the general availability $Avail_{i,p}$ of the scheduler i for the priority p as (Equation B.5):

$$Avail_{i,p} = \begin{cases} 1 & \text{if } OutOfWork_i = 1 \\ RedAvail_{i,p} \times \dfrac{|FullSchedulers| \times AvgHRed_i}{\sum\limits_{k \in FullSchedulers} AvgHRed_k} & \text{otherwise} \end{cases} \tag{B.5}$$

First, if the scheduler i waited for work, then it was available for actors of any priority. If not, then the general availability of the scheduler i is calculated as its reduction based availability times the ratio of the historical average number of reductions executed (times a scaling factor *FullSchedulers*) and the average of every other scheduler.

Having defined the availability of a scheduler, we can now finally define $MigLim_{i,p}$, the migration limit for the scheduler i, and run-queue with priority p as (Equation B.6):

$$MigLim_{i,p} = \left\lfloor \sum_{j=1}^{n} MaxRQSize_{j,p} \times \frac{Avail_{i,p}}{\sum_{k=1}^{n} Avail_{k,p}} \right\rfloor \tag{B.6}$$

The migration limit is, therefore, the proportion of the number of actors (based on the run-queues maximum lengths) distributed to every scheduler and priority based on the general availability ratio for each one of them. If all the actors in the system have the same priority (the usual case) and every run-queue has the same availability, then this formula is equal to the average maximum sizes of all run-queues.

This intricate formula deals with historic values in order to smooth variations in the migration limits. The rationale is that a strict current situation analysis could lead to abrupt variations on the migration limits. These variations could cause many actor migrations (and occasionally actor bouncing) and therefore could hurt performance. By taking into consideration the historic values and average reduction-based availabilities of each scheduler, the migration impact of big local variations on the number of actors can be reduced while at the same time guaranteeing a load-balancer with soft real-time properties.

B.2 Migration Paths

When we compare the definition of the migration paths to that of the migration limits, we quickly realize that it is much simpler. After the migration limit for each run-queue is established, for each queue and priority we calculate the difference $D_{i,p} = MaxRQSize_{i,p} - MigLim_{i,p}$. Independently for each priority, we then sort the run-queues using $D_{i,p}$ as the sorting key. Run-queues with negative D values are chosen for immigration while positive values for emigration of actors.

In the first phase, the migration paths are determined by taking the run-queue with the maximum D value and setting the migration path towards the run-queue with the minimum D value, then setting the immigration of the run-queue with the second highest D value towards the run-queue with the second lowest value, and so on. This process is repeated until all the remaining run-queues have negative D values, positive D values, or there are no more run-queues.

In the second phase, run-queues are associated in reverse order. For example, if the remaining run-queues have negative D values, then the sources of immigration are chosen from the queues with positive D values in decreasing value of D. Figure B.1 depicts this process for both cases, assuming, for the sake of simplicity, that all run-queues have the same migration limit.

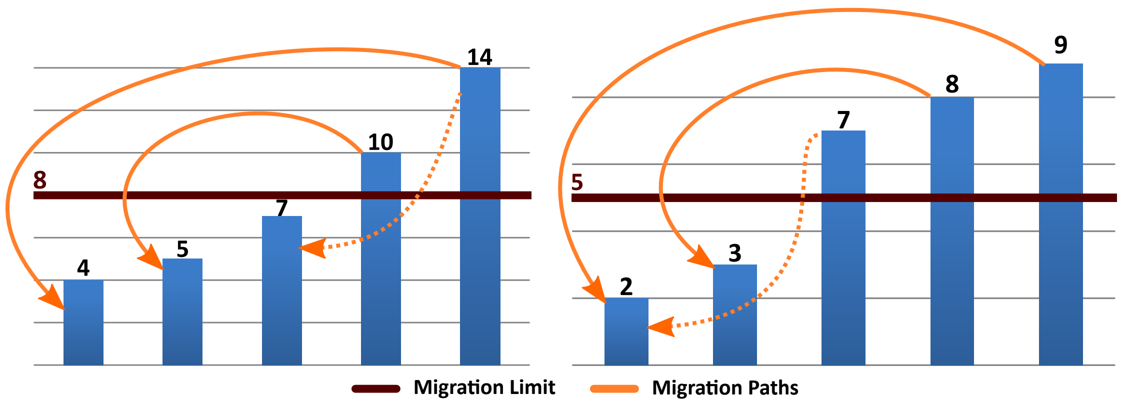


Figure B.1: Migration paths determination. On the left the case in which more run-queues with negative *D* values exist. On the right, more run-queues with positive *D* values. Dashed arrows indicate paths determined on the second phase, *i.e.*, run-queues for which a migration path could not be determined in the first phase.

APPENDIX C

Altix UV 2000 Preliminary Experimental Evaluation

An experimental evaluation of a large NUMA machine, the Altix UV 2000 platform, was also performed. This platform possesses 24 nodes and 192 cores. Table C.1 summarizes its general characteristics. This machine has hyper-threading capable processors, however all of our tests were run with hyper-threading disabled.

Each one of the 24 nodes of the Altix UV 2000 platform (Figure C.1) has an

Altix UV 2000	
NUMA Nodes	24
Cores	192
Frequency	2.40 GHz
Total RAM	756 GiB
L3 Cache	20 MiB
NUMA Factor	6.6 to 10.0
Linux Kernel	3.0.74
GCC	4.3.4

Table C.1: Altix UV 2000 platform specification

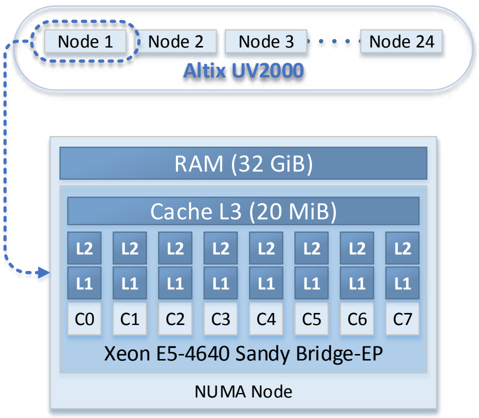


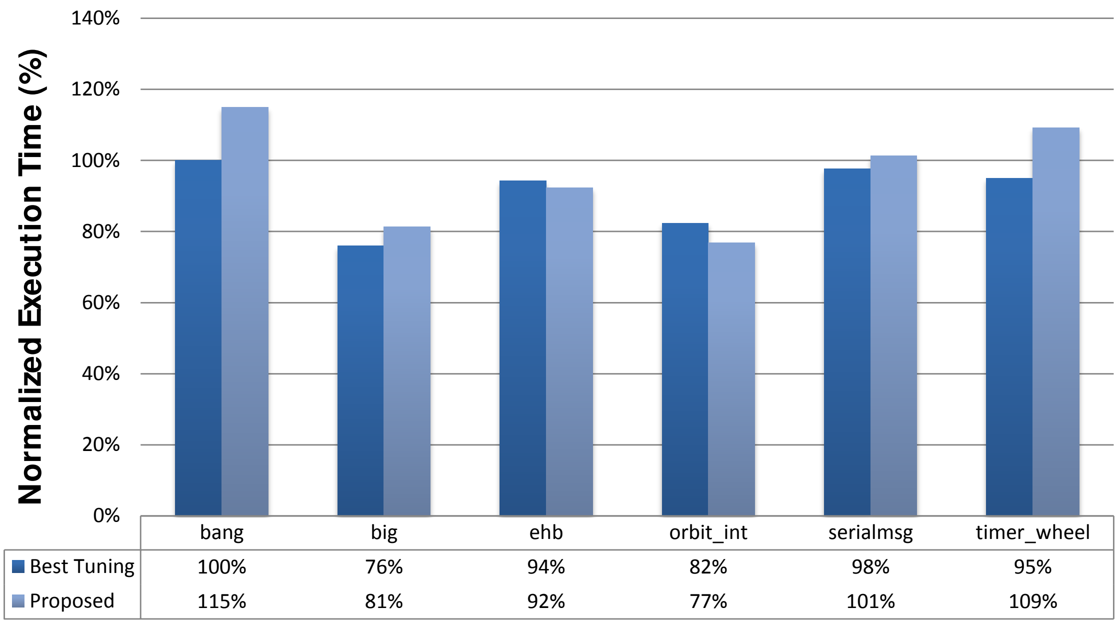
Figure C.1: Simplified architectural view of the Altix UV 2000 experimental platform

eight-core Xeon E5-4640 Sandy Bridge-EP processor. The NUMA nodes are connected through SGI's proprietary NUMAlink6 (bidirectional) in a partial cube topology (shown in Figure 2.3). Each link has a bandwidth of 6.7 GiB/s per direction. Additionally, nodes are organized in groups of four. Nodes in the same group share extra links between them, improving intra-group communication performance. Furthermore, NUMA nodes are physically arranged in pairs, one pair per blade on the rack. Nodes on the same blade have special interconnections between them. The performance effects these different interconnections create are shown in Figure 2.4.

C.1 Experimental Results

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 C.2 depicts the performance comparison between the default and modified configurations and Table C.2 shows the full list of the achieved speedups.

This platform, however, requires further analysis. The execution times for both



	bang	big	ehb	orbit_int	serialmsg	timer_wheel
■ Best Tuning	100%	76%	94%	82%	98%	95%
■ Proposed	115%	81%	92%	77%	101%	109%

Figure C.2: Normalized execution time of the benchmarks for the short input sizes on the Altix UV 2000 platform.

	Default	**Best Tuning**
bang	0.87	0.99
big	1.23	0.93
ehb	1.08	1.02
orbit_int	1.30	1.07
serialmsg	0.98	0.95

Table C.2: Altix UV 2000 speedups for the proposed configuration

the original and the modified VMs are much higher than we expected. Using the *original* unmodified VM, we measured the execution times of the benchmarks using the short datasets on both NUMA 32 and Altix UV 2000 platforms. Table C.3 shows the obtained results.

	NUMA 32	**Altix UV 2000**
bang	0.89 s	110.48 s
big	0.43 s	3.23 s
ehb	1.01 s	11.38 s
orbit_int	1.14 s	8.59 s
serialmsg	2.25 s	412.27 s
timer_wheel	1.92 s	117.62 s

Table C.3: Benchmarks Execution Time

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 not only has better processors but it has six times more processors and a better NUMA interconnection than the NUMA 32 platform. Yet, its performance is still much worse.

However, the strange behavior we see on the Altix UV 2000 platform is not exclusive to the Erlang VM. During the analysis of the execution of a simple application written in C [CFNM13], a naïve brute-force parallel traveler salesman problem (TSP) solver[1], we realized that the execution times were very high. We compared the execution times of the Altix UV 2000 platform to that of a laptop (Intel Core i7-2630QM, at 2.0 GHz) for the same input size (17). Altix UV 2000 took 65 s to solve the problem, while the processor on the laptop took 3.7 s.

[1]The code is available at `https://github.com/mbcastro/tsp`.

We soon realized that the problem involved some form of false-sharing. All of the program's data structures already had their members padded to avoid being on the same cache line. Yet, this small padding (64 B) was not enough to ensure performance on the Altix UV 2000 platform. In fact, to achieve it, we had to increase the padding all the way up to a full memory page size (4 KiB). After this modification we were able to execute the application in 3.5 s. We suspect this behavior to be related to the cache-coherence protocols used by the Altix UV 2000 platform.

Similar to the TSP application, the Erlang VM code has several padded data structures. As one might expect, it uses as padding size the system's cache line size. If the same problem is happening to the Erlang VM, only a comprehensive set of changes to the its codebase would solve this issue.

C.2 The NUMA Bottleneck

As in any platform, to achieve good performance, 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 original unmodified 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[2]. 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 communication were not a bottleneck, we would see performance

[2]We used the `numactl` [48] Linux utility to restrict the access to PUs and memory of the VM to just one NUMA node.

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 C.4 show exactly the opposite: the restricted execution has a speedup of up to 16.6 when compared to the performance of the regular execution.

	NUMA 32	Altix UV 2000
`bang`	16.6	4.5
`serialmsg`	1.3	1.6

Table C.4: Measured Slowdown

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 interconnection 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 on the NUMA interconnection 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 interconnection such as that 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.

APPENDIX D

Extended Abstract in French

Environnements d’Exécution à Base d’Acteurs pour Plate-formes Multi-cœurs à Mémoire Partagée Hiérarchique

Emilio de Camargo Francesquini

Table des matières

D.1 Introduction

La croissante demande pour des processeurs puissants a fait les fabricants de puces utiliser dans leurs solutions des concepts qui sont une combinaison de force brute et d'innovation. L'augmentation de la taille de la mémoire cache des processeurs, du parallélisme au niveau des instructions et de la fréquence d'horloge ont été, au cours des dernières décennies, leurs principales approches pour accomplir cette mission. Cependant, ces approches semblent avoir atteint un point où elles, en eux-mêmes, ne suffisent pas à assurer la courbe d'amélioration de performance prévue par la loi de Moore et attendue par les consommateurs. Une augmentation exponentielle de la consommation d'énergie liée à une augmentation linéaire de la fréquence d'horloge [BBS+00] et une plus grande complexité pour concevoir de nouveaux processeurs, sont des raisons qui ont changé la façon dont la poursuite de processeurs à chaque fois plus performants est faite.

En 2004, quand le premier processeur multi-cœur pour le marché grand public a été dévoilé, on pouvait déjà se rendre compte de la nouvelle tendance dans les stratégies de développement des processeurs pour les années suivantes [Lar09]. Avec l'avènement des processeurs multi-cœurs, les concepteurs de matériel ont pu maintenir les améliorations de performance attendues et, en même temps, garder la conception des processeurs à un niveau gérable de complexité. Par conséquent, l'amélioration de la performance est devenue autant un problème de logiciel comme elle était, jusque-là, un problème matériel. Bien que ce nouveau paradigme pourrait faire face à une certaine résistance [Sut05, 1] [1], les processeurs multi-cœurs se sont généralisés à la fois sur les environnements professionnels et personnels. Même les processeurs mobiles à faible consommation d'énergie, tels que Samsung Exynos 5 Octa [2] et NVIDIA Tegra 4 [NVI13], ont déjà plus de quatre cœurs alors que certains processeurs multi-cœurs tels que la puce Kalray MPPA-256 ont jusqu'à 256 cœurs [ABB+13, DML+13].

Bien que l'offre de processeurs multi-cœurs soit abondante, des solutions faciles et simples pour le développement d'applications parallèles n'existent pas. Les outils traditionnels tels que les *mutex* et les moniteurs ont toujours été une source de problèmes

[1] Dans ce document, nous distinguons les citations des publications scientifiques de ressources en ligne. A cet effet, deux principaux formats de citation sont utilisés. Les initiales des auteurs avec l'année pour les publications scientifiques et la numérotation séquentielle pour les ressources en ligne.

(tels que les *race conditions*, les interblocages, la famine, et le non-déterminisme) pour les développeurs. Plusieurs solutions ont été proposées pour tenter résoudre ce genre de problèmes. La plupart d'entre eux vise à faciliter l'utilisation efficace du matériel tout en protégeant le développeur de ses particularités. Ces solutions peuvent être classées dans les catégories suivantes :

- **APIs bas niveau** Ce sont généralement des fonctions de bas niveau du système d'exploitation (SE) qui créent de nouveaux processus ou threads et qui donnent accès à des instructions au niveau du matériel comme *test-and-set*, *fetch-and-add* et leurs variantes. Dans cette catégorie de solutions le développeur de l'application est responsable de la synchronisation entre les différentes lignes d'exécution. Pour l'accomplir, on utilise généralement des *mutexes*, sémaphores ou moniteurs qui sont mis à disposition par l'API du SE. L'un des représentants les plus connus de cette catégorie est l'API POSIX Threads [But97].

- **Services du système d'exploitation** Contrairement à l'API bas niveau, les solutions de ce type sont des services de haut niveau offerts par le SE. Parallax OS [MS11] ainsi que le GCD de Mac OS X [3], également disponible sur OpenBSD [4], sont quelques exemples. En particulier, la solution de GCD est constituée de plusieurs files d'attente d'exécution, chacune avec une priorité différente, qui sont mises à disposition de l'application. L'application peut, à son tour, mettre des travaux dans la file d'attente pour l'exécution et être notifiée dès leur achèvement. Si les tâches de l'application peuvent être divisées en plusieurs bloc petits parts pour être mis dans la file d'attente et exécutés par le SE, ce modèle se traduit par une bonne utilisation des capacités parallèles de la machine.

- **Intergiciels, *Frameworks* et Langages de Programmation** Les solutions de cette catégorie peuvent être très distinctes en ce qui concerne leurs approches à l'exécution parallèle. Deux grands exemples de'Intergiciels/frameworks sont OpenMP et MPI [Qui04]. Bien qu'ils peuvent sembler être mutuellement exclusives, ces deux solutions particulières sont souvent utilisés ensemble, par exemple, dans un cluster composé de plusieurs machines multi-cœurs :

OpenMP pour la parallélisation local et MPI pour la distribution entre les machines.

Les approches basées sur les langages de programmation s'efforcent de cacher les détails de la machine (et parfois du SE) du développeur de l'application. Quelques exemples sont Charm++ [KK93] et Cilk [BJK+95], qui peuvent être vus comme des extensions parallèles aux langages C/C++. Il existe également des modèles de langages basés sur le modèle d'acteur comme Erlang [Arm10] et Scala [OAC+04], et les langages basés sur les concepts de de mémoire transactionnelles tels que Clojure [Hal09]. Non seulement ces solutions offrent une vue de haut niveau des lignes distinctes de l'exécution, mais elles fournissent également une méthodologie de programmation, c'est-à-dire, elles influencent l'architecture de l'application. Les applications écrites dans ces langages sont généralement capables de prendre avantage de manière transparente des nouvelles machines (même celles avec de nombreux cœurs). Dans ce cas, l'environnement d'exécution (RE– *Runtime Environment*) devient le responsable de la répartition du travail entre les cœurs disponibles, même s'ils ne sont pas tous locaux.

L'un des principaux inconvénients des approches de bas niveau est que le développeur de l'application doit connaître, à l'avance, la topologie du réseau et/ou l'architecture de la machine sur laquelle l'application va être exécutée, sinon des pénalités de performance pourraient s'ensuivre. Si les informations de plate-forme ne sont pas disponibles à l'avance, les développeurs d'applications ont besoin d'écrire plusieurs routines pour gérer l'exécution pour qu'elle soit capable de s'adapter au comportement de l'application et aux particularités soit celles de la machine réelle ou celles du SE. Notre avis est que, sauf lorsque c'est strictement nécessaire, le développeur de l'application doit se soucier des détails concernant l'application et ne pas se soucier des caractéristiques du matériel ou du SE. Pour cette raison, dans ce travail, nous allons nous concentrer sur la catégorie de solutions comprenant *Intergiciels,* Frameworks *et Langages de Programmation*.

La plupart des solutions de cette catégorie ont déjà une sorte d'optimisation qui tiennent compte de la plate-forme matérielle. Nous en discutons plus dans la Section D.5. Cependant, il y a peu ou pas de travail du tout, dirigé à l'exécution efficace d'environnements basés sur le modèle d'acteur sur de grandes plates-formes multi-cœurs.

Le modèle d'acteur est présent dans plusieurs systèmes critiques, tels que ceux qui soutiennent WhatsApp [5, 6] avec 430 millions d'utilisateurs (avec plus d'un million de connexions simultanées) et Facebook Chat [7, 8] avec plus d'un milliard d'utilisateurs (le nombre réel de connexions simultanées n'a pas été rendu publique). Ces systèmes servent des milliers de clients en même temps, donc exigeant des ressources informatiques importantes habituellement fournis par plates-formes basées sur multiprocesseurs et multicœurs. Les architectures NUMA (Section D.2) représentent une part importante de ces plates-formes. Pourtant, peu de recherches ont été menées sur l'adéquation des environnements d'exécution basés sur le modèle d'acteur à ces machines. Les REs actuels supposent un espace mémoire plat, donc ils ne sont pas aussi performants que possible. Les plates-formes NUMA présentent plusieurs défis pour les REs basés sur le modèle d'acteur dans des domaines de la gestion de la mémoire, ordonnancement et l'équilibrage de charge. Dans ce document, nous analysons et caractérisons des applications basées sur ce modèle et, à la lumière de ce qui précède, proposons des améliorations à ces REs.

Comme preuve de concept, nous avons appliqué nos idées dans un RE réel, la machine virtuelle (VM) d'Erlang. Erlang a une VM *open-source* dédiée qui utilise l'approche basée sur événements. Le langage a été créé sur la base du modèle d'acteur d'origine avec quelques adaptations mineures et, en tant que tel, il a une syntaxe propre et cohérente pour la gestion des acteurs. De plus, nous croyons qu'Erlang est un bon choix de langage parce qu'il est proche de la description du modèle original d'acteur, les conclusions que nous tirons de cet environnement peuvent être presque directement appliquées à d'autres langages basés sur les acteurs tels que Salsa [VA01]. Cette VM modifiée tire parti des caractéristiques NUMA et de la connaissance de l'application pour une meilleure gestions de la mémoire, de l'ordonnancement et des décisions d'équilibrage de charge.

D.1.1 Contributions

Le modèle d'acteur a été initialement proposé dans le cadre de l'intelligence artificielle [HBS73] et seulement quelques années plus tard il a également commencé à être considéré comme un modèle pour la concurrence [Agh86]. Dans ce modèle, chaque flux d'exécution distinct est considéré comme un acteur. Un acteur peut créer d'autres acteurs et aucune donnée est partagée. La seule façon d'observer ou de modifier l'état d'un acteur est l'envoi ou la réception d'un message. Pour communiquer, chaque acteur possède une boîte aux lettres privée. Les messages peuvent être envoyés à un autre acteur qui, à leur tour, les traite de manière asynchrone, à sa convenance et pas nécessairement dans l'ordre de réception. Si un acteur n'a aucun message à traiter, il peut suspendre son exécution et garder ce status jusqu'à ce qu'un délai soit atteint ou un nouveau message soit remis dans la boîte aux lettres. La livraison des messages est indépendante de l'état de l'acteur, c'est-à-dire, même si l'acteur est occupé, le dépôt de nouveaux messages n'est pas bloqué.

L'isolement de la mémoire, l'échange de messages et le traitement en série des messages par chaque acteur permet la non-existence de verrous, de sémaphores ou tout autre outil de synchronisation spécifique. La synchronisation réelle entre les acteurs est atteinte grâce à l'échange de messages. Bien que puissant, cette abstraction - volontairement - cache du développeur de l'application les particularités architecturales de la machine. Ainsi, le RE devient le responsable d'une utilisation efficace de l'architecture sous-jacente.

Les acteurs ont certaines caractéristiques qui les distinguent des autres modèles de programmation. Par exemple, la durée de vie d'un acteur est généralement très courte. Les acteurs sont créés pour effectuer des tâches très spécifiques et puis ils disparaissent. De plus, les acteurs sont souvent créés en quantités beaucoup plus importantes que le nombre de cœurs de la machine. La raison derrière cela est double. Tout d'abord, les acteurs sont, en général, inactifs par la plupart du temps puisque pour la plupart de leurs vies, ils attendent des messages. Deuxièmement, l'état global d'un système est conservé par l'ensemble des acteurs ; il n'y a pas de mémoire partagée, de sorte que, afin d'accéder aux données qui doivent être partagées entre de nombreux acteurs, par exemple, on a besoin de créer un acteur qui tient cette valeur. Le grand nombre d'acteurs et leur indépendance rendent le modèle d'acteur un bon choix pour profiter des nouvelles machines multi-cœur.

Les architectures NUMA sont une tendance forte sur des machines avec un grand nombre de cœurs. Cela a été motivé, en part, par le fait que ces architectures sont capables d'exécuter des applications régulières développées pour machines avec un espace de mémoire plat sans aucune modification. Cependant, si leurs caractéristiques matérielles spécifiques ne sont pas prises en compte, la concurrence pour les ressources partagées peut entraîner une dégradation importante de performance [KCS04, TASS07]. Comme nous le montrons dans la Section D.5, il existe des méthodes d'utilisation efficaces pour les plates-formes NUMA mais, au mieux de nos connaissances, il n'y a pas de recherches publiées traitant de l'adaptation de REs basés sur le modèle d'acteur pour ces plates-formes.

Afin de s'attaquer à ce problème, nous avons décidé de diviser notre approche en deux fronts distincts. Nous allons d'abord étudier comment les applications basées sur acteurs se comportent et, avec cette connaissance, nous allons adapter et évaluer des REs existants actuellement.

► **Comportement des Applications Basées sur Acteurs** Notre avis est qu'afin d'améliorer les performances des applications d'acteurs et de leurs REs, nous devons d'abord comprendre le comportement de ces applications. Il s'agit, en fait, de la première contribution de notre travail. Nous analysons certaines applications et nous montrons que non seulement l'acteur moyen est de courte durée (et que ceux qui sont de longue durée sont la plupart du temps inactifs), mais aussi qu'il existe un type particulier d'acteur que nous appelons *hub*. Les hubs sont des acteurs impliqués dans beaucoup plus de communications que l'acteur moyen. Ils sont, en fait, les orchestrateurs de l'exécution de l'application. Nous définissons l'ensemble des acteurs qui échangent des messages avec un acteur hub *le groupe d'affinité du hub*. Nous présentons ensuite une heuristique exploitant les comportements des hubs qui utilise le fait que les communications inter-groupe d'affinité sont rares et que l'acteur présent sur le groupe d'affinité d'un hub a été, probablement, créé par ce même hub.

► **Approche Hiérarchique** Grâce à une meilleure compréhension du comportement de ces applications, notre prochaine contribution est la proposition d'un ensemble d'optimisations pour un RE basé sur le modèle d'acteur. Ces optimisations ont comme but l'amélioration de l'utilisation des ressources matérielles par la correspondance de l'exécution des applications à la plate-forme matérielle sous-jacente. Nous introduisons le concept de *home nodes* (essentiellement c'est le nœud NUMA où le tas de l'acteur est placé) et expliquons comment on peut améliorer les performances des applications d'acteurs sur les plates-formes de mémoire hiérarchique. Puis, en utilisant l'heuristique développée par la contribution précédente et les home nodes, nous proposons une série d'améliorations aux équilibreurs de charge et aux politiques de placement initial d'acteurs sur ces REs.

En introduisant nos idées dans une VM moderne, la VM Erlang, nous avons pu mesurer et évaluer l'efficacité de notre proposition. Nous décrivons ce travail avec les résultats de performance obtenus dans différentes plates-formes pour des applications réelles et synthétiques créées spécifiquement pour tester les différents aspects de l'exécution.

D.1.2 Contexte Scientifique

Cette thèse a été développée dans le cadre d'une co-tutelle entre l'Université de São Paulo (USP) et de l'Université de Grenoble. Elle a été divisée en trois phases, la première et la dernière développées à São Paulo, à l'Institut de Mathématiques et de Statistiques (IME) de l'USP, et la deuxième à Grenoble, au Laboratoire d'Informatique de Grenoble (LIG).

La première phase comprenait la formation de base du candidat, ainsi que l'enquête et la création d'un sujet de recherche. Ce travail a été réalisé dans le groupe de systèmes distribués (GSD) de l'IME. Au cours de cette phase, le candidat a reçu un financement provenant des projets Baile[2] et CHOReOS[3]. Baile est un projet de recherche financé par HP Brésil et réalisée à l'IME-USP FLOSS Competence Center en collaboration avec HP Labs. CHOReOS fait partie du projet de recherche du programme européen FP7, développé à l'IME-USP FLOSS Competence Center ainsi que plusieurs institutions européennes. Ces projets ont permis une étroite collaboration avec des chercheurs de HP Labs, et plusieurs universités et entreprises européennes.

La deuxième phase comprend la période pendant laquelle la plupart des recherches contenues dans ce document ont été effectuées. C'est également au cours de cette phase que la publication des premiers résultats a été faite (vous pouvez consulter la liste complète dans l'Appendice A). Le candidat a travaillé avec l'équipe NanoSim

[2]Baile : Enabling Scalable Cloud Service Choreographies – `http://ccsl.ime.usp.br/baile/`. Grant HP-037/11.

[3]CHOReOS - Large Scale Choreographies for the Future Internet – url http ://choreos.eu/. FP7-ICT-2009-5, Grant # 257178

(Nanosimulations and Embedded Applications for Hybrid Multi-core Architectures). Les membres du projet NanoSim ont eu l'occasion de collaborer avec des chercheurs des institutions de recherche du monde entier tels que le *Universidade Federal do Rio Grande do Sul* (UFRGS, Brésil), *Commissariat à l'Energie Atomique et aux Énergies Alternatives* (CEA, France), *Institut National de Recherche en Informatique et en Automatique* (INRIA, France) et l'Université de Yaoundé 1 (Cameroun). Pendant cette période, l'auteur a reçu un financement de projet CAPES/COFECUB[4].

La troisième phase concerne la conclusion de la thèse et la publication des résultats finaux ainsi que quelques résultats satellites. Elle a été faite avec le GSD et concerne aussi la présentation de nos résultats dans conférences internationales. Les commentaires reçus dans ces conférences, ainsi que le travail que nous avons développé, ont permis la consolidation des résultats sous la forme présentée dans ce document. Cette dernière phase a été financée par une bourse CAPES (*bolsa institucional*) accordée à l'auteur.

Cet appendice contient une version résumée du texte la thèse. Son organisation suit globalement celle du texte principal. Vous pouvez consulter la thèse pour en savoir plus sur les sujets abordés dans cet appendice. La correspondance entre les sections de cet appendice et les chapitres du texte principal est indiquée ci-dessous.

► La Section D.2 présente une révision rapide des concepts, des outils impliqués dans l'utilisation efficace des multi-cœurs et des plates-formes multi-cœurs hiérarchiques et le modèle d'acteur y compris des détails sur les REs et les applications existantes (Chapitre 2). Ensuite, nous décrivons notre travail sur le comportement des applications d'acteurs. D'abord, nous caractérisons les applications puis présentons une heuristique comportementale basée sur ces résultats (Chapitre 3).

► S'appuyant sur les résultats décrits par la section précédente, la Section D.3 (Chapitre 4) décrit notre proposition d'un RE qui prend en compte les caractéristiques hiérarchiques de la machine.

► Dans la Section D.4 (Chapitre 5), nous montrons notre méthodologie d'évaluation, y compris les plates-formes utilisées, les modifications que nous avons faites à la VM Erlang et nos résultats expérimentaux.

[4]CAPES/Cofecub Projet #660/10

- La Section D.5 (Chapitre 6) traite de plusieurs ouvrages liés non seulement à l'optimisation des REs d'acteur aux plates-formes de mémoire hiérarchiques, mais aussi à l'optimisation des REs parallèles en général.
- La Section D.6 (Chapitre 7) apporte notre conclusion, y compris une vue d'ensemble de nos contributions, et expose nos perspectives.

Dans ce document, nous avons divisé les références bibliographiques dans deux catégories différentes. Dans la première catégorie, nous énumérons les publications traditionnelles tels que les documents scientifiques et académiques. Les clés pour la citation utilisées pour ces travaux sont définis en utilisant les initiales des auteurs et l'année de publication. La deuxième catégorie a été utilisée pour les ressources en ligne telles que les *home pages* de produits, des manuels de logiciels, du code source et des communiqués de presse. Dans ce cas, les clés de citations sont simplement numérotées séquentiellement par leur ordre d'apparition dans le texte.

D.2 Motivations et les Principes de Base

Les langages de programmation parallèle sont maintenant à la mode. Ces langages ne sont pas nouveaux [AV90, IKBW$^+$79]. Cependant, jusqu'à il y a une vingtaine d'années, ils ont été, dans la pratique, limités aux activités de recherche. Au cours des dernières années, principalement en raison de l'introduction de processeurs multi-core, on observe un regain d'intérêt pour ces langages, notamment pour ceux fondés sur le modèle d'acteur [KSA09]. Ces langages permettent à leur utilisateurs d'utiliser la puissance des nouveaux processeurs multi-core d'une façon directe et efficace. Les langages basés sur le modèle d'acteur sont utilisés dans plusieurs domaines différents, la plupart du temps pour construire des serveurs de haute disponibilité. Certains utilisateurs sont très connus comme Amazon, Facebook, Yahoo et Twitter [CT09, 9].

Les processeurs multi-core ont marqué l'adoption générale des niveaux de caches partagés. Les grands caches partagés simplifient le fonctionnement de ces processeurs, cependant ils peuvent également entraîner une contention importante sur le cache, et des variations imprévisibles dans le temps d'exécution. En outre, certaines architectures ont une hiérarchie asymétrique de mémoire, donc des coûts de communication entre les PUs, même entre cœurs dans le même processeur, ne sont pas constants. Les architectures NUMA augmentent encore l'asymétrie de la hiérarchie de la mémoire par l'ajout de mémoire locale à chaque nœud. Le RE du modèle d'acteur, comme une couche supplémentaire sur le système d'exploitation, dispose d'informations additionnelles sur le comportement de l'application. Ces informations peuvent être utilisées par le RE pour prendre de meilleures décisions d'ordonnancement et d'équilibrage de charge.

Dans cette section nous présentons une brève introduction sur le modèle d'acteurs et sa relation avec les machines NUMA. Ensuite, nous décrivons le fonctionnellement d'un RE pour le modèle d'acteur.

D.2.1 Le Modèle d'Acteur et les Plates-formes NUMA

Certains aspects du modèle d'acteur peuvent être directement utilisés pour améliorer les performances des REs disponibles actuellement. La connaissance du comportement des acteurs, leur graphe de communication, les coûts de communication de la machine cible et la relation entre les acteurs sont des exemples de connaissances disponibles dans le RE qui peuvent être utilisées pour améliorer ses propres décisions d'ordonnancement et d'équilibrage de charge.

Pour illustrer certains de ces aspects, nous utilisons une application réelle comme exemple. Cette application, CouchDB [18], est une base de données NoSQL écrite en Erlang. Nous pensons que c'est un bon exemple pour illustrer les aspects de l'application qui nous intéressent. Les résultats de cette section ont été obtenus en utilisant CouchDB version 1.3.0 sur la machine décrite dans la Section D.4.

D.2.1.1 Durée de Vie des Acteurs

Les acteurs ne sont pas égaux. Chaque application a des acteurs spécialisés qui mènent plusieurs types différents de travail. Il serait un exercice futile d'essayer de lister tous les types possibles d'acteurs. Nous pouvons, cependant, définir deux grandes catégories d'intérêts et d'analyser leurs propriétés générales. Dans ce contexte, nous avons étudié les acteurs à courte et à longue durée.

Notre exemple d'application, CouchDB, crée de nombreux acteurs de courte durée. En fait, 99,5% des acteurs vivent moins de 1,5 s et 88,9% moins de 0,1 s. Le temps nécessaire pour créer un acteur (minimal) est, en moyenne, de 1,5 μs.

La proportion réelle entre les acteurs à courte et à longue durée est spécifique à chaque application. Nous pouvons, cependant, tirer quelques conclusions sur le RE de cet simple exemple. Tout d'abord, le RE doit être très efficace pour l'exécution des acteurs de courte durée, sinon les applications telles que CouchDB n'auraient pas de bonnes performances. Pour les acteurs de courte durée, la décision de l'emplacement initial de l'acteur, c'est-à-dire, le choix sur quel cœur l'acteur va être exécuté, doit être rapide. Deuxièmement, l'ordonnanceur doit être capable de traiter de grandes quantités d'acteurs et aussi traiter la création d'acteurs en rafales. Le modèle MapReduce [DG08], utilisé par CouchDB, Riak [PJS12] et beaucoup d'autres applications basées sur des acteurs, fait exactement cela : il crée de nombreux acteurs de courte durée dans un court laps de temps. Troisièmement, un acteur typique de courte durée est (ou vas essayer d'être) actif pendant son exécution. Néanmoins, le nombre d'acteurs vivants dans l'application généralement dépasse le nombre de cœurs disponibles sur la plate-forme, ce qui fait une solution de partage de temps une partie essentielle du RE.

D.2.1.2 Coûts de Communication

En principe, toutes les communications sur le modèle d'acteur sont basées sur le passage de messages. Comment le RE le fait dépend de l'implémentation. Cependant, il est raisonnable de supposer qu'une implémentation efficace sur une plate-forme SMP utiliserait mémoire partagée pour fournir ce service. Les coûts de communication en utilisant la mémoire partagée sur des machines NUMA sont définis non seulement par la taille du message, mais également par l'emplacement de l'émetteur et du récepteur. Dans les plates-formes NUMA, les coûts de communication peuvent facilement devenir l'un des facteurs déterminants de la performance des applications [CPRA+12b]. La Figure D.1 montre la pénalité de performance associée à l'envoi des messages de différentes tailles en utilisant le coût d'échange entre des acteurs sur la même PU (c'est-à-dire, des acteurs partageant le premier niveau de cache) comme la ligne de base. Pour les petits messages, les performances inter-nœud peuvent être plus de sept fois plus lentes tandis que pour les grands messages la performance est d'environ la moitié de celle de la ligne de base. Pour le cas intra-nœud, la performance est entre deux et trois fois inférieure entre des petits et des grands messages.

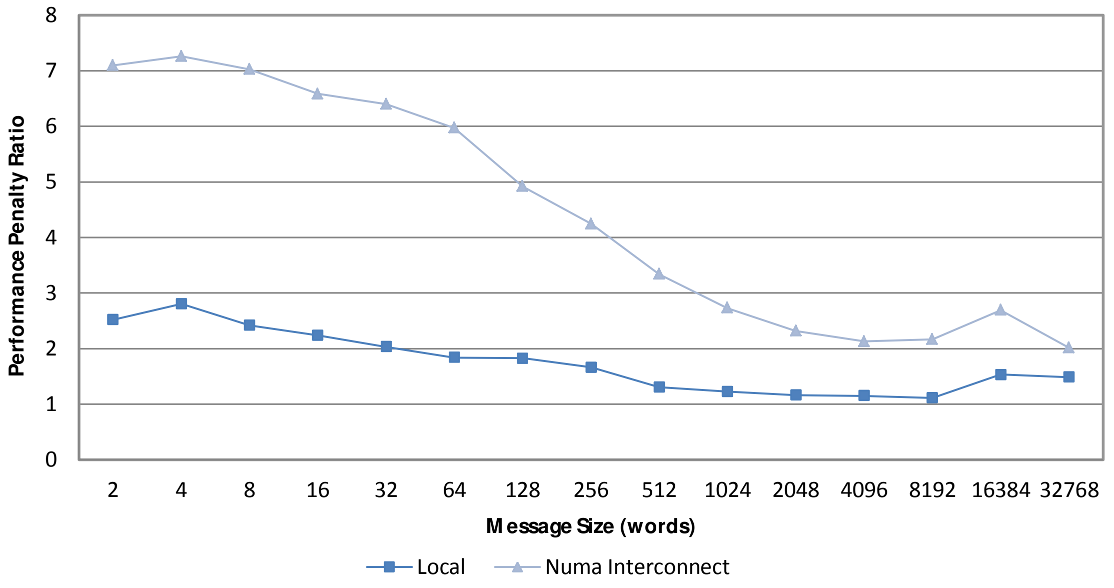


FIGURE D.1: Rapport entre les coûts de communication intra et inter-nœud associés à l'échange des messages de différentes tailles. La ligne de base représente le temps nécessaire pour la transmission des messages entre acteurs qui se trouvent sur le même cœur. Des essais ont été effectués sur la plate-forme décrite dans la Chapitre D.4. Les intervalles de confiance (95 %) sont trop petits pour être visibles sur cette figure, donc ils ne sont pas tracés.

Les acteurs qui ont un flux intense de communication entre eux et qui ne sont pas placés de manière optimale, peuvent provoquer des effets indésirables. Au-delà de la durée plus longue de la communication, ces effets peuvent inclure, par exemple, la contention sur les interconnexions matérielles telles que les liens NUMA et l'augmentation du nombre de défauts de cache.

Afin de montrer un de ces effets, nous avons créé une application artificielle simple (représentée sur la figure D.2). Cette application a été conçue pour démontrer l'impact d'un mauvais placement sur le cache du processeur. Lors de l'exécution, nous avons soigneusement choisi les acteurs communicants et délibérément nous les avons placés aussi près que possible (sur le même cœur) et nous avons comparé ce placement au deuxième meilleur placement (cœur distinct, même socket). Cette migration minimale a causé environ un millier de fois plus défauts de cache que le placement optimal.

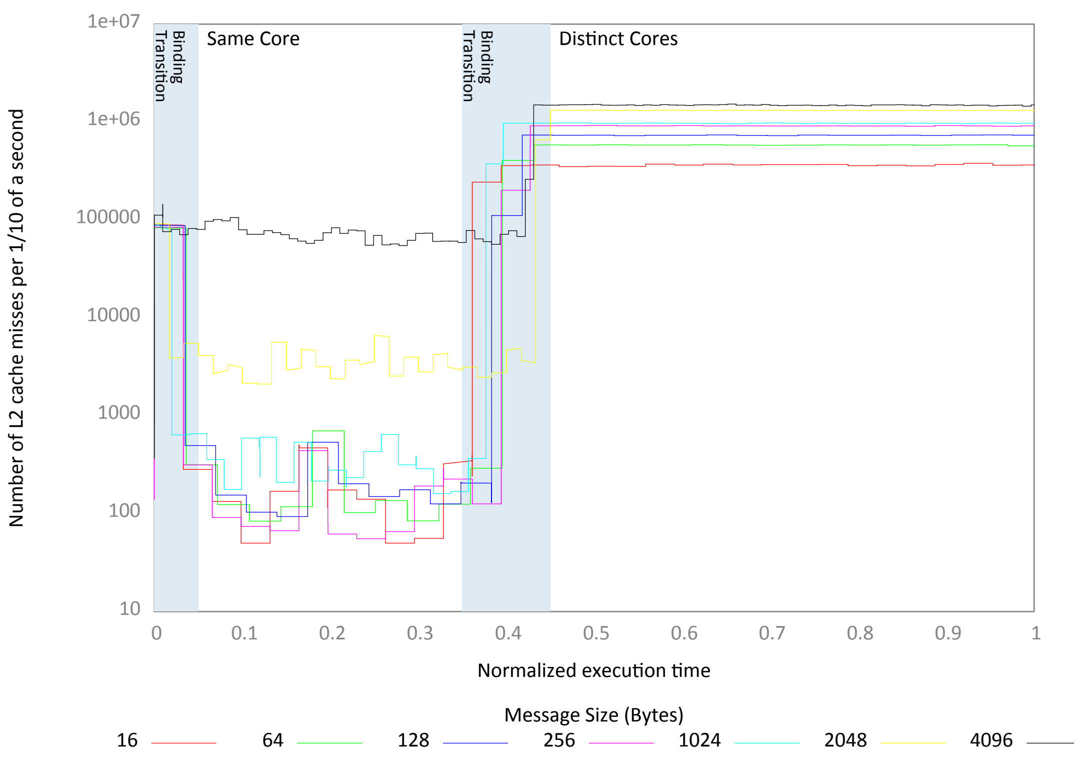


FIGURE D.2: Nombre de défauts de cache L2 par 1/10 de seconde causées par l'envoi de messages de différentes tailles à l'aide de deux ensembles distincts de PUs. Comme le temps nécessaire pour exécuter l'application dépend de la taille des messages, l'axe horizontal a été normalisé de 0 à 1 pour représenter le début et la fin des exécutions. Les régions ombrées (de 0 à 0.05 et de 0.35 à 0.45) marquent la transition entre placements.

D.2.1.3 Hubs

Les acteurs ont généralement une fonction définie à leur création. Certaines de ces fonctions sont naturellement plus demandées que d'autres, ce qui rend la communication et la distribution de charge non uniforme. Nous appelons *hubs* les acteurs qui échangent, de manière significative, plus de messages que l'acteur moyen et qui se communiquent avec un grand nombre d'acteurs différents. Nous définissons l'ensemble des acteurs qui échangent des messages avec un acteur hub comme son groupe d'affinité.

Pour illustrer ces définitions nous avons analysé le graphe de communication de

CouchDB. Le graphe illustré par la Figure D.3 est une représentation des communications qui ont eu lieu lors d'une exécution réelle de cette application.

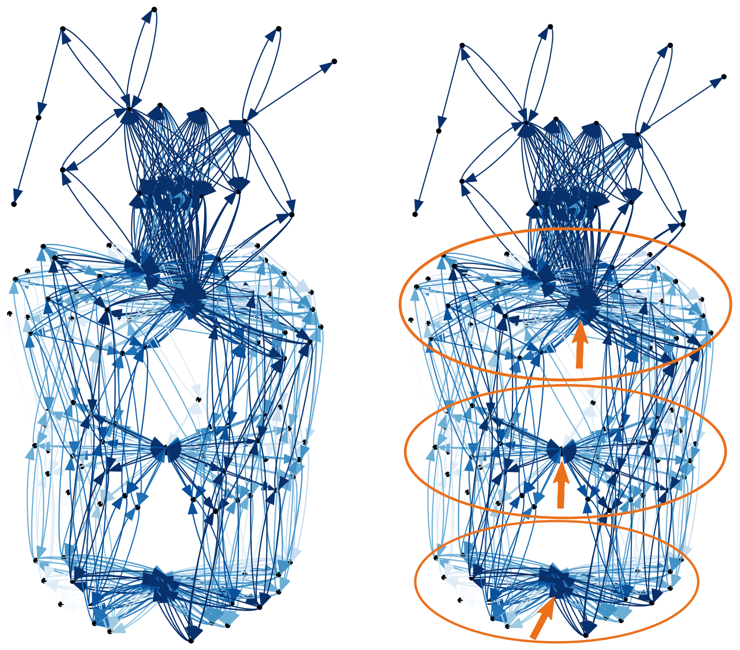


FIGURE D.3: Le graphe de communication de CouchDB. Les vertex représentent les acteurs et les arêtes des communications entre eux. Les acteurs qui ont communiqué au moins une fois sont liés. Plus les arêtes sont foncées, plus le nombre de messages échangés a été important. Sur la gauche, la totalité du graphe est représentée. Sur la droite, le même graphe avec quelques hubs et leurs groupes d'affinité sont mis en évidence.

Les informations sur les hubs et leur groupe d'affinité sont disponibles au RE pendant l'exécution de l'application. Ainsi, lors de la décision des migrations, l'équilibreur de charge pourrait prendre le groupe d'affinité d'un hub en considération. L'objectif serait de réduire les coûts de communication entre le hub et son groupe d'affinité et donc d'améliorer la performance globale du système.

D.2.2 Les Environnements d'Exécution à Base d'Acteurs

Le développement d'un RE pour le modèle d'acteurs doit prendre en compte plusieurs aspects du modèle d'acteur original tels quels l'échange asynchrone de messages, des boîtes aux lettres privées, la création dynamique d'acteurs et l'absence de données partagées. Dans cette section nous décrivons ces choix avec un intérêt particulier pour le RE du langage Erlang qui a été modifié pour créer le prototype utilisé dans notre évaluation expérimentale.

D.2.2.1 Ordonnancement d'Acteurs

Les REs basés sur le modèle d'acteur sont généralement construits en utilisant l'une des deux approches : l'approche basée sur *threads* et l'approche basées sur événements. La principale différence est que dans la première approche chaque acteur est représenté par un *thread* ou un processus du système d'exploitation. Dans la seconde approche chaque acteur est représenté par une structure de données interne du RE. Dans ce cas c'est le RE, et pas le système d'exploitation, le responsable de l'ordonnancement des acteurs et d'équilibrage de charge. Même si cela rend le RE plus complexe, il le rend également plus performant car le RE a la possibilité d'effectuer des optimisations d'exécution qui n'auraient pas été possibles autrement.

L'approche basée sur événements est le choix fait par le RE du langage Erlang, alors que Scala, par exemple, donne au développeur la possibilité de choisir entre les deux [OAC+04]. Erlang représente chaque acteur comme une structure de données simple. C'est pourquoi une application Erlang typique, qui a des dizaines (parfois des milliers) d'acteurs, peut être exécutée efficacement, même sur des machines qui n'ont que quelques PUs.

Un ordonnanceur d'acteurs comme celui d'Erlang fonctionne en créant un *thread* du système d'exploitation pour chaque PU disponible sur le système (Figure D.4). Ces *threads* sont appelés *ordonnanceurs*. Même si l'association des ordonnanceurs aux PUs pourrait améliorer la performance à travers d'une meilleure utilisation des caches du processeur, ils ne sont pas associés par défaut.

Chaque ordonnanceur a une file d'exécution qui maintient les acteurs exécutables qui lui sont assignées. Un acteur est exécutable quand il n'est pas dans l'attente pour messages ou de toute autre opération bloquante. Quand l'ordonnanceur choisit un acteur pour l'exécution, l'acteur sera exécuté jusqu'à ce que son quota prédéterminé du processeur s'épuise ou il soit bloqué par une opération d'E/S. À ce moment, l'acteur sera préempté par l'ordonnanceur et remis sur la file d'attente. L'ordonnanceur ensuite prend le prochain acteur sur la file d'attente pour exécution.

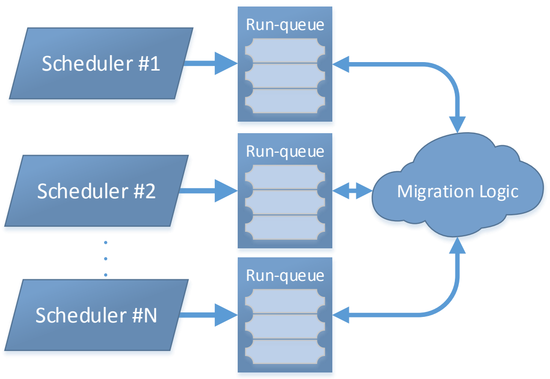


FIGURE D.4: Les ordonnanceurs et files d'attente dans un processeur multi-core avec N PUs. Habituellement, il y a autant d'ordonnanceurs que le nombre de PUs disponibles. Chaque ordonnanceur a sa propre file d'exécution et est responsable de l'exécution des acteurs dans cette file d'attente. Si un ordonnanceur devient oisif parce que sa file s'est vidée, il procède à l'exécution d'un algorithme de vol de travail. Il existe également un équilibreur de charge qui est exécuté périodiquement pour assurer qui les files d'attente soient bien équilibrées.

Au cours de l'exécution de l'application, les tailles des files d'attente de chaque ordonnanceur peuvent devenir très différentes les unes des autres. Même si les files d'attente ont été équilibrées au début de l'exécution, chaque acteur a une durée de vie distincte. En outre, quand un nouvel acteur est crée, il est mis dans la même file d'attente d'exécution de son créateur et les acteurs n'ont pas les mêmes comportements quand il s'agit de la création de nouveaux acteurs.

Pour contrôler ce déséquilibre, les environnements d'exécution utilisent deux stratégies : vol de travail et équilibrage de charge périodique. Si la file d'attente d'un ordonnanceur est vide, il va commencer une logique de migration pour voler des acteurs des autres ordonnanceurs. Ce mécanisme serait suffisant pour garder tous les ordonnanceurs occupés (en supposant qu'il y a assez de travail pour tous). Cependant, le vol de travail n'est pas en soi suffisant pour garantir que chaque acteur reçoive une part juste d'utilisation de PUs. C'est pourquoi le RE exécute périodiquement une routine d'équilibrage de charge entre les files d'attente. Les critères utilisés pour déterminer les migrations ne comprennent pas ni les groupes d'affinité ni si l'acteur est un hub. Seul la taille des files d'attente et la position de l'acteur sont prises en considération.

Il y a un autre type de déséquilibre de charge qui est typique dans les systèmes sous-chargés. Dans ce cas, une stratégie appelée compactage de charge est utilisée. Cette stratégie utilise aussi peu que possible les ordonnanceurs et essaie de minimiser la fréquence dans laquelle les ordonnanceurs deviennent oisifs. La raison derrière cela est qu'un petit nombre d'acteurs dans le système augmente la tendance des acteurs à rebondir entre les ordonnanceurs car toute variation dans le nombre d'acteurs pourrait provoquer une migration, par exemple, par l'équilibreur de charge. La compactage de charge fonctionne en détectant la fréquence dans laquelle les ordonnanceurs deviennent oisifs et si cette fréquence est supérieure à un seuil prédéfini, le RE migre les acteurs exécutables à un ensemble plus restreint d'ordonnanceurs actifs. Les ordonnanceurs restants sont suspendus. Si, à un certain moment, les ordonnanceurs actifs ne sont pas suffisants pour gérer tout le travail, quelques-uns des ordonnanceurs en suspension sont réveillés et la charge est rééquilibré. La raison pour laquelle le RE a besoin de ce genre de comportement est liée à la façon dont les acteurs sont traités. Les acteurs sont censés être indépendants de la plate-forme, et par conséquent, il n'y a pas de sens dans la liaison entre l'exécution d'un acteur (ou un groupe d'acteurs) à un ordonnanceur. D'autre part, le RE a des informations qui peuvent être utilisées pour éviter ce genre de comportement errant, par exemple, en essayant de garder les hubs et leurs groupes d'affinité proches évitant ainsi des rebondissements.

D.2.2.2 Gestion de Mémoire

Les acteurs ne partagent pas de données, et c'est exactement la façon dont quelques REs, tels quel la machine virtuelle Erlang, sont construits. Dans ce RE chaque acteur possède son propre tas. Cela rend plus facile à mettre en œuvre un *garbage collector* efficace car un tel collecteur n'a pas besoin d'arrêter la machine virtuelle, dans la mesure où il suffit d'arrêter l'acteur sur lequel il travaille. En fait, plusieurs acteurs de courte durée jamais ne passent par un cycle du *garbage collector* au cours de leur vie. Ces acteurs sont normalement complètement éliminés une fois qu'ils ont terminé leurs exécutions.

Comme le tas de chaque acteur est indépendant, l'échange de messages se fait en copiant les données. C'est-à-dire, chaque message est copié depuis le tas de l'acteur expéditeur vers le tas de l'acteur récepteur. Il y a quand-même une exception dans la machine virtuelle Erlang, le type *binary*. Les binaires plus gros que 64 octets sont attribués dans un tas binaire partagé et messages qui contiennent des objets de ce type sont envoyés par référence plutôt que par valeur.

L'allocation de tas est une tâche intrinsèque liée à la création d'un nouvel acteur. Le tas d'un acteur est attribué par l'ordonnanceur de l'acteur créateur. Ce qui signifie que l'ordonnanceur responsable de l'exécution de l'acteur créateur est également responsable de l'allocation et de la copie des paramètres vers le tas de l'acteur créé. Dans les machines avec un espace-mémoire plat, l'emplacement de la mémoire allouée ne varie pas, il est toujours local. D'autre part, sur des machines NUMA, le système d'exploitation peut utiliser plusieurs politiques différentes pour le placement de la mémoire. Linux, par exemple, utilise par défaut une politique de *first-touch*. Pour le RE Erlang cela signifie que l'emplacement du tas de l'acteur sera le nœud où l'ordonnanceur qui l'a créé exécutait. Nous allons appeler cet endroit le *home node* de l'acteur.

C'est important de se rendre compte que les home nodes ne sont pas définitifs. Prenez par exemple un acteur qui, pour une raison quelconque, a été migré. Lors de son exécution, il peut être nécessaire d'agrandir son tas pour s'adapter à de nouvelles données. Souvent, il n'est pas possible d'allouer de la mémoire supplémentaire en utilisant la même adresse mémoire et, dans ce cas, une copie du tas complet vers le nouvel emplacement doit être fait. Si le nouvel ordonnanceur vers lequel l'acteur a été migré n'est pas sur le même nœud NUMA que le home node de l'acteur, son home node sera changé et toutes les fonctionnalités du RE qui dépendent de cette information devront être mises à jour. Par ailleurs, le coût d'une simple opération de croissance du tas qui aurait été proportionnelle à la taille de la mémoire sur une machine avec un espace de mémoire plat dépendra de l'emplacement et du home node de l'acteur affecté.

Certains des choix pris par les développeurs des REs actuels suggèrent qu'ils n'ont pas été écrits en considérant les particularités des architectures NUMA. Ce problème n'est pas spécifique à la machine virtuelle Erlang. Au meilleur de notre connaissance, aucun RE ne prend en considération les aspects NUMA de la plate-forme sous-jacente. Dans la section suivante, nous discutons certaines considérations que le RE pourrait employer pour mieux s'adapter aux plates-formes NUMA.

D.2.3 Conclusions

Récemment le modèle d'acteur a eu une importante croissance dans sa popularité. Une des raisons derrière cette croissance est à la fois la disponibilité de processeurs multi-core et aussi la difficulté qui est communément associée à la programmation de systèmes concurrents et parallèles avec des outils traditionnels tels que POSIX Threads. Ce modèle est basé sur de simples exchanges de messages et pour cela il se présente comme un substitut plus simple à l'utilisation. Ce modèle apporte aussi d'autres avantages, comme par exemple, la distribution transparente et la modélisation [SJ11] et vérification [TKL+12] d'applications.

Les REs actuels sont très optimisés pour les architectures SMP. Ces optimisations ont atteint un point où le développeur d'applications a rarement besoin de se soucier avec les détails de la plate-forme sous-jacente. Cependant, comme le modèle d'acteur gagne des adeptes, ces applications commencent a tourner sur des machines plus puissantes qui sont fréquemment des machines avec une architecture NUMA. Les applications qui tournent sur ce type d'architecture n'atteignent pas donc la performance attendue.

Dans cette section, nous avons brièvement décrit les architectures NUMA le modèle d'acteur et la façon dont il est actuellement employé. Nous avons montré quelques détails qui peuvent être pris en considération pour améliorer la performance de ces ER dans ce contexte. Dans la section suivante, nous allons explorer ces détails avec plus d'attention afin de proposer un ensemble d'améliorations inspiré par ces observations.

D.3 Une Approche Hiérarchique pour les Environnements d'Exécution à Base d'Acteurs

Les plates-formes NUMA présentent des défis non seulement pour les REs basés sur le modèle d'acteur mais aussi à toutes les applications concurrentes. Les coûts distincts pour accéder à différentes parties de la mémoire entraînent un nombre considérable de problèmes qui, entre autres, impliquent dans le choix de placement de processus et de mémoire, l'ordonnancement, l'équilibrage de charge et la migration de mémoire. Nous sommes intéressés par des moyens d'exploiter efficacement ces plates-formes en utilisant les REs actuellement disponibles avec quelques modifications. Pour le faire, nous avons analysé le comportement de certaines applications créées en utilisant le modèle d'acteur. Plus précisément, nous avons étudié leur graphe de communication et le comportement de leurs hubs.

Nous étions à la recherche de motifs communs à l'exécution de ces applications et l'analyse des graphiques de communication a abouti à deux conclusions principales. Tout d'abord, les hubs sont généralement responsables de la création de la majorité des acteurs qui appartiennent à son groupe d'affinité (cette heuristique va être beaucoup utilisée dans notre approche). Deuxièmement, le graphe de la communication et, par conséquent, le groupe d'affinité des acteurs, sont extrêmement dynamiques. Essayer de maintenir une représentation en ligne du graphe ou du groupe d'affinité pourrait apporter une surcharge importante au RE. Nous proposons donc une approche plus simple basée sur quelques *hints* fournis par le développeur de l'application.

Les développeurs ont souvent de bonnes indications sur les caractéristiques d'exécution de l'application. Ils peuvent, par conséquent, fournir des *hints* sur les acteurs qui sont possiblement des hubs. Les *hints* ne changent pas le comportement fonctionnel des applications et le RE pourrait, à sa discrétion, ne pas les tenir en compte. Toutefois, le RE peut également les utiliser pour l'aider à prendre de meilleures décisions. Notre approche fonctionne en créant les outils de développement qui permettent le développeur effectuer le marquage des acteurs qui sont, selon lui, des hubs. Cela peut être fait au cours de la création de l'acteur, ce qui signifie que le développeur a, au moment de la création, certains éléments de preuve qui indiquent que l'acteur sera un hub. Ce genre de preuve peut également apparaître lors de l'exécution. Une décision pendant l'exécution signifie probablement qu'elle dépend de l'évaluation des données qui seront disponibles uniquement lors de l'exécution. Par exemple, les acteurs choisis par les algorithmes des élections en ligne pourraient devenir des hubs au cours de l'exécution de l'application et donc ils peuvent changer leur comportement après leur création.

Notre proposition est basée sur deux aspects principaux du RE, la politique d'équilibrage de charge et le soutien des affinités des acteurs. L'équilibrage de charge vise non seulement à maintenir chaque PU disponible occupée la plupart du temps, mais aussi d'assurer que chaque acteur obtienne une part équitable du temps des PUs. Le soutien des affinités des acteurs essaie de garder les acteurs, et leur groupe d'affinité, toujours proches de sorte que la communication entre eux soit efficace. Parfois, ces deux objectifs peuvent entrer en conflit. Par exemple, l'affinité maximale serait de placer tous les acteurs sur la même PU. Pourtant, cela laisserait les PUs restantes oisives minimisant ainsi l'équilibre de charge. Nous cherchons un bon compromis, en termes de performance, entre ces deux aspects de l'exécution.

De manières très différentes pour résoudre le problème d'équilibrage de charge sont utilisées par les REs. Les REs basés sur threads délèguent généralement la solution au système d'exploitation. Une solution naturelle étant donné que chaque acteur est un thread du système d'exploitation. D'autre part, les RE basées sur événements n'ont pas ce choix. Il faut qu'ils le résolvent eux-mêmes, en utilisant normalement les files d'attente d'exécution simples ou multiples. La version d'une unique file d'attente fonctionne essentiellement par l'emploi d'un *pool* de threads qui consomme travail de cette file. Cette version n'a pas besoin d'une logique d'équilibrage de charge séparée. D'autre part, la version avec multiples files d'attente utilise une file distincte pour chaque thread. Dans ce cas, comme le comportement de chaque acteur est différent, il pourrait y avoir un certain déséquilibre entre les files. C'est pourquoi les algorithmes vol de travail et d'équilibrage de charge sont employées.

Le modèle basé sur threads impose certaines limites à ce que l'on est capable d'observer et agir puisque les décisions d'équilibrage de charge sont prises par le système d'exploitation. Nous nous concentrerons donc sur l'approche basée sur événements. Parmi les approches basées sur événements, une seule file d'attente est l'option la plus simple. Elle fonctionne très bien dans une machine d'espace mémoire plat avec un petit nombre de PUs. Cependant, conforme le nombre de cœurs d'exécution grandit, la contention pour accéder à la file d'attente commune augmente, ce que limite l'évolutivité du système [Lun08]. En outre, sur une plate-forme NUMA, les threads seront probablement distribués partout dans la machine. Dans ce scénario, la file d'attente commune distribuerait l'exécution des acteurs uniformément à travers des threads. Cela va provoquer un rebondissement des acteurs entre les threads, la création d'un nombre important de défauts de cache et donc l'augmentation du trafic sur l'interconnexion NUMA. En d'autres termes, cette solution ne favorise pas la *soft-affinity*. D'autre part, les approches avec plusieurs files d'attente avec une file par thread, à condition que les fils soient associés aux PUs, ont la soft-affinity assurée. Cependant, cette approche doit prendre en compte le déséquilibre éventuel entre les files d'attente. C'est à ce moment que le vol de travail et des algorithmes d'équilibrage de charge sont mis en place. Un système chargé tend à avoir un petit nombre de migrations préservant donc la soft-affinity. Lorsque ce n'est pas le cas, les algorithmes de compression de charge essayent d'éviter les migrations en diminuant le nombre d'ordonnanceurs actifs à un minimum. Ces raisons nous incitent à croire qu'un RE basé sur événements avec multiples files d'attente soit la solution la plus appropriée pour un RE basé sur le modèle d'acteur sur une plate-forme de NUMA.

Notre proposition de maintenir à la fois l'équilibrage de charge et l'affinité des acteurs est centrée autour des mécanismes d'équilibrage de charge du RE. En appliquant l'heuristique décrite au début de cette section, nous pouvons modifier les algorithmes de placement d'acteurs et de migration d'une façon qui rendre possible faire les deux choses à la fois. L'approche est divisée dans les catégories complémentaires qui nous présentons maintenant.

D.3.1 Placement Initial d'Acteurs

Plusieurs politiques de placement initial sont possibles quand on considère le comportement attendu d'un acteur. Proportionnellement, les hubs exigent beaucoup plus du RE que leurs homologues réguliers. Ils sont généralement aussi parmi les plus grands créateurs d'acteurs dans une application. Ainsi, il est logique d'essayer de disperser les hubs d'une manière que minimise la concurrence pour les ressources. D'autre part, les acteurs réguliers sont plus susceptibles de communiquer dans leur groupe d'affinité, il est donc logique de les placer à proximité de leurs hubs. Nous proposons donc l'utilisation de deux politiques de placement initial différentes, une pour les hubs et d'autre pour les acteurs réguliers. Les hubs devraient être distribués par la machine, alors que les acteurs réguliers doivent être placés près de leur hub/groupe d'affinité, sur le même nœud NUMA si possible. La meilleure façon de faire cette distribution de hubs dépendent du comportement de l'application. Par exemple, nous pourrions privilégier communication en plaçant les hubs proches, mais pas sur la même PU (*compact*), ou privilégier l'indépendance des ressources en plaçant les acteurs autant distants que possible (*spread*). Ces deux stratégies favorisent une bonne répartition initiale des hubs entre les cœurs disponibles de la machine.

D.3.2 Équilibrage de Charge et Vol de Travail Hiérarchiques

Au cours de l'exécution de l'application, les déséquilibres iront se produire même avec une bonne politique de placement initial. C'est pourquoi le RE a besoin d'un équilibreur de charge périodique. En outre, si une file d'attente devient vide entre les tours d'équilibrage de charge, une solution de vol de travail pourrait être utilisée comme une solution légère temporaire afin de conserver les PUs occupées. Les deux algorithmes vont migrer acteurs entre les files d'attente gérées, toutefois, afin d'améliorer la performance globale du système sur une plate-forme NUMA, la façon dont un candidat est choisi pour être migré est importante. Les mesures que nous proposons pour le choix d'un candidat de migration sont, d'abord, la migration des acteurs pour les rapatrier. Si cela n'est pas suffisant, nous essayons de régler la déséquilibrage en migrant les acteurs entre les ordonnanceurs dans leurs home nodes. Seulement si ces mesures ne suffisent pas, l'algorithme envisage les acteurs restants pour la migration inter-nœud. Ces étapes visent à maintenir et rétablir la proximité entre les acteurs dans le même groupe d'affinité tout en les maintenant près de leur home node et donc de leur tas.

D.4 Évaluation Expérimentale

Des environnements d'exécution d'acteurs optimisés pour les architectures SMP sont déjà la norme. D'autre part, malgré le fait que la mémoire partagée des plates-formes multi-core hiérarchiques soit présente dans de nombreux systèmes, au mieux de nos connaissances, des optimisations qui prennent en compte les caractéristiques distinctes des plates-formes NUMA sont très limitées. Prenant en considération les observations et propositions présentées dans las sections précédentes, nous avons modifié le code source d'un RE populaire et réel, la machine virtuelle Erlang. Dans cette section nous présentons la mise en ouvre et l'évaluation expérimentale de notre proposition pour un RE sur les architectures NUMA.

Dans cette section nous présentons l'architecture qui a été évaluée, les modifications que nous avons faites sur la machine virtuelle Erlang, les benchmarks et applications utilisés et, finalement, nous présentons l'analyse des résultats expérimentaux.

NUMA 32	
NUMA Nœuds	4
Cœurs	32
Fréquence	2.27 GHz
RAM Totale	64 GiB
L3 Cache	24 MiB
Facteur NUMA	1.2 to 3.6
Noyau Linux	3.5.7
GCC	4.7.2

TABLE D.1: Spécifications de la plate-forme NUMA 32

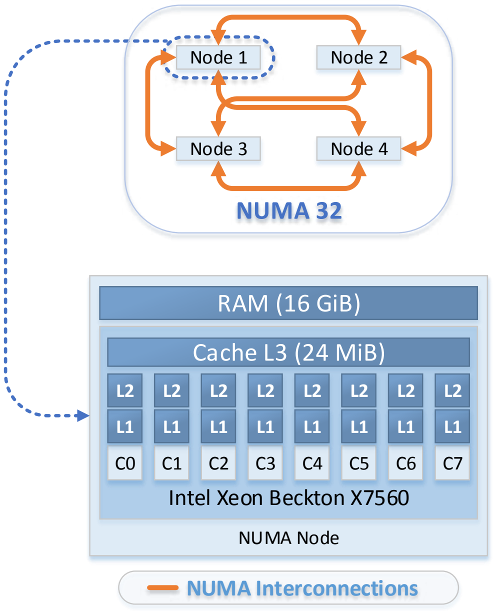


FIGURE D.5: Diagramme simplifié de la plate-forme expérimentale NUMA 32

D.4.1 Plate-forme Expérimentale

Afin d'évaluer les modifications proposées, nous avons utilisé une machine NUMA (NUMA 32) composée par quatre nœuds, chacun avec 8 PUs, donc 32 PUs au total. Le Tableau D.1 résume ses caractéristiques générales et la Figure D.5 montre un diagramme architectural simplifié. Cette plate-forme possède des processeurs avec la technologie *hyper-threading*, pourtant tous nos tests ont été réalisés avec la technologie *hyper-threading* désactivée pour éviter les interférences pendant la mesure des performances.

La plate-forme NUMA 32 est composée de quatre nœuds, chacun avec un processeur Intel Xeon X7560 Beckton de huit cœurs. Le LLC de chaque processeur est partagé par tous ses cœurs et l'interconnexion NUMA est un graphe complet.

D.4.2 La Machine Virtuelle Erlang Modifiée

La machine virtuelle Erlang dispose de plusieurs outils d'exécution intéressants qui peuvent être utilisés pour faire le mises au point des applications et aussi pour les profiler sur les plates-formes NUMA. Cependant, pour effectuer l'analyse que nous présentons dans ce document et pour réaliser notre proposition d'un RE capable d'exécuter de manière efficace sur une plate-forme NUMA, nous avons eu besoin d'adapter et d'étendre la machine virtuelle actuelle. Dans cette section nous décrivons brièvement ces modifications.

Nous avons utilisé la machine virtuelle Erlang version R15B02 comme la base des nos modifications. Pour le reste de ce texte, nous ferons référence à la machine virtuelle Erlang non-modifiée comme *originale* et à notre propre machine virtuelle customisée comme *modifiée*. La plupart des changements que nous avons introduits dans le code sont indépendants de plate-forme. Néanmoins, certaines d'entre eux (surtout ceux liés à l'utilisation des APIs NUMA) sont spécifiques du système d'exploitation Linux. Par conséquent tous nos tests ont été effectués sur ce SE. En dépit de cela, nous croyons que nos modifications sont suffisamment génériques pour être facilement portées à un autre système d'exploitation tels que FreeBSD et Windows. Maintenant nous soulignons les changements les plus importants que nous avons mis en place.

► **Hubs Tracking** Une partie importante de notre proposition est le marquage des hubs. C'est la raison pour laquelle nous avons introduit quelques modifications pour le rendre possible. Ces marquages prennent la forme de *hints*. Les *hints* peuvent être donnés comme un paramètre supplémentaire lors de la création de l'acteur ou par l'utilisation d'un *flag* en temps d'exécution. Avec ces informations, la VM modifiée sera capable d'associer l'exécution des hubs à une PU spécifique et ainsi éviter leur migration en raison de l'équilibrage de charge ou vol de travail.

L'interface que nous avons créée est montré dans le Listing D.1. Comme base de comparaison, les lignes 2-3 montrent la création d'un acteur régulier. Lignes 5-6 montrent la création d'un acteur hub à l'aide de la nouvelle interface. Enfin, la Ligne 8 montre comment signaler un acteur existant comme une hub et la Ligne 10 montre comment supprimer ce flag.

► **Politiques de Migration** Nous avons modifié les algorithmes utilisés par la machine virtuelle Erlang pour effectuer le vol de travail et l'équilibrage de charge. Nous avons introduit dans ces algorithmes les idées qui ont été présentées dans la Sec-

```erlang
% Creation d'un acteur commun
Pid1 = spawn_opt(A_Module, A_Function, FunctionArgs,
    []).
% Creation d'un hub
Pid2 = spawn_opt(A_Module, A_Function, FunctionArgs,
    [hub_process]).
% Effectue le marquage de l'acteur comme hub
erlang:system_flag(hub_process, true).
% Retire le marquage de l'acteur comme hub
erlang:system_flag(hub_process, false).
```

Listing D.1: Interfaces en Erlang pour la création et le marquage de hubs

tion D.3. Maintenant l'utilisateur de la VM peut choisir, en temps d'exécution, la stratégie de migration souhaitée. Les stratégies disponibles sont `default`, `numa`, et `disabled`. La première stratégie correspond au comportement habituel de la VM. La deuxième stratégie comprend nos modifications tels que la migration hiérarchique et la migration locale avant une migration globale. La stratégie `disabled` désactive tous les mécanismes de migration et est très utile pour le débogage.

► **Politiques de Placement Initial** La machine virtuelle originale a comme comportement par défaut le placement des acteurs créés dans le même ordonnanceur de leurs créateurs. Ce comportement a été modifié et, maintenant, c'est possible de choisir la politique de placement initial indépendamment pour les hubs et les acteurs réguliers. Les politiques de placement initiaux disponibles sont : `default`, `compact`, `scatter`, `circular`, et `random`.

► **Allocation et Initialisation du Tas** Dans une plate-forme NUMA, l'allocation de mémoire est plus complexe qu'une allocation dans une plate-forme SMP. Pour cette raison nous avons modifié la VM Erlang pour que l'allocation de mémoire soit toujours locale, même si l'ordonnanceur choisi par la politique de placement initial ne se trouve pas sur le même nœud de l'ordonnanceur où l'acteur créateur se trouve. L'option de faire l'allocation locale peut être faite pendant l'initialisation de la VM. Si l'allocation du tas d'un acteur doit être faite dans un nœud non-locale, les données doivent être copiées depuis le nœud d'origine (créateur) vers le nœud d'exécution. Cette copie prend du temps, et pour cela nous avons introduit un paramètre qui donne le choix du moment où la copie doit être faite, pendant la création (option par défaut) ou pendant le premier ordonnancement de l'acteur (*deferred allocation*).

D.4.3 Benchmarks et Applications

Pour évaluer les performances de notre prototype, nous avons utilisé la suite de benchmarks BenchErl [APR+12]. BenchErl a des tests qui évaluent différents aspects de la VM Erlang. Des tests qui évaluent des APIs spécifiques ou qui sont *CPU-bound* (tels que ceux qui testent les *ETS tables* et la fonction `erlang:now/0`) ont été retirés de notre évaluation car ils ne sont pas pertinents pour les aspects que nous voulons évaluer, c'est-à-dire, ceux où la communication et le placement des acteurs ont un rôle important. Nous avons légèrement modifié le code de référence. Notre modification a été limitée à l'ajout des *hints* nécessaires pour informer le RE sur les hubs. Nous décrivons brièvement les benchmarks choisis ci-dessous. Nous avons aussi évalué la performance de deux applications réelles. Ses descriptions avec celles des benchmarks sont dans le Tableau D.2.

D.4.4 Résultats Expérimentaux

Nous avons testé la version modifiée de la VM de façon extensive sur la plate-forme NUMA 32. Nous montrons aussi nos résultats préliminaires sur la plate-forme Altix UV 2000 dans l'Appendice C, une machine avec un nombre de nœuds important dans laquelle ni la VM original ni la VM modifiée ont été capables d'exécuter de manière efficace.

Nous montrons d'abord les résultats obtenus avec les benchmarks et après les résultats obtenus avec les applications réelles. Pour une analyse plus approfondie des résultats, vous pouvez consulter le Chapitre 5 qui contient aussi des discussions sur l'efficacité de nos modifications.

D.4.4.1 Benchmarks

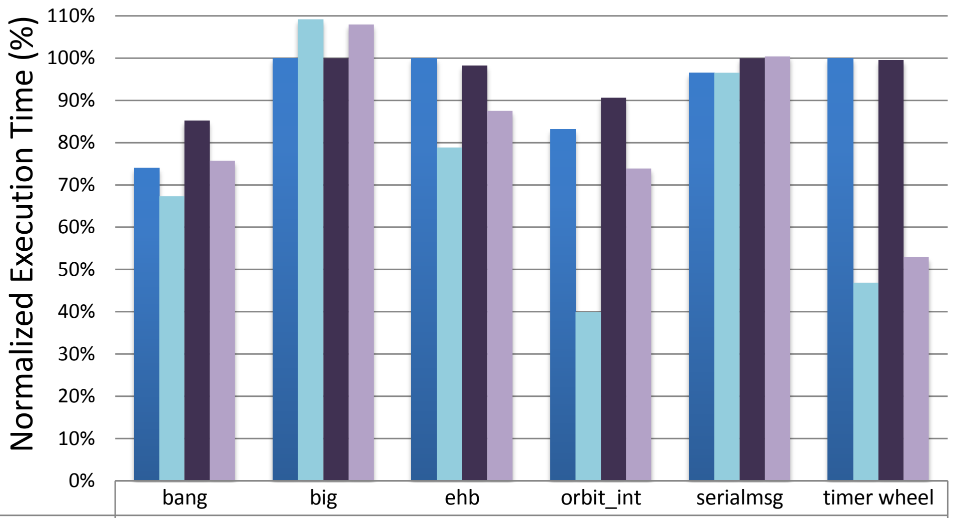
La Figure D.6 montre le temps d'exécution normalisé des benchmarks choisis, par rapport à la configuration par défaut (c'est-à-dire, nous prenons comme référence

TABLE D.2: Benchmarks et applications utilisés dans l'évaluation de performance

Benchmark/Application	**Description**
Bang	Échange de message depuis plusieurs acteurs vers un
Big	Échange de message depuis plusieurs acteurs vers plusieurs
Ehb	Version écrite en Erlang du benchmark Hackbench [27] pour stresser les ordonnanceurs
Orbit_int	Implémentation en Erlang d'une table de hachage distribuée avec une architecture maître/esclave.
Serialmsg	Échange de message depuis plusieurs acteurs vers un et un vers plusiers
Timer_wheel	Échange de message depuis un acteur vers plusieurs avec du temps limite
Sim-Diasca	Simulateur d'événements discrets [SKZ+11, 26], avec la simulation *City Waste Management* [Bou13]
ErlangTW	Simulateur d'événements discrets [TDM12] qui utilise l'abordage *Time Warp synchronization protocol* pour la simulation. Les tests ont été effectués en utilisant la simulation PHOLD [Jon86]

la performance de la VM sans paramètres optionnels). Nous montrons des résultats pour deux tailles d'entrée de données : courtes et intermédiaires. Tous les résultats présentés dans cette section ont été obtenus à partir des moyennes d'au moins 30 échantillons et analysées en utilisant des intervalles de confiance de 95 % pour assurer la signification statistique. Comme il n'y a pas beaucoup de variation dans les données, les intervalles de confiance sont trop petits pour être représentés graphiquement. Pour cette raison, ils ont été omis de la figure. Par exemple, l'intervalle de confiance (95 %) du temps d'exécution moyen pour `ehb` avec la configuration proposée et la taille d'entrée courte est $0.663 \pm 0.002\,53$ s, ce qui représenterait une variation normalisée de moins de 0.33 % dans la Figure D.6.

Dans cette plate-forme nous avons amélioré la performance de quatre benchmarks : `bang` , `ehb`, `orbit_int` et `timer_wheel`. Le Tableau D.3 contient la liste complète de speedups qui ont été obtenus avec la configuration proposée en relation aux configurations par défaut et *best tuning*.



	bang	big	ehb	orbit_int	serialmsg	timer wheel
🟦 Short - Best Tuning	74%	100%	100%	83%	97%	100%
🟦 Short - Proposed	67%	109%	79%	40%	97%	47%
■ Intermediate - Best Tuning	85%	100%	98%	91%	100%	100%
🟪 Intermediate - Proposed	76%	108%	88%	74%	100%	53%

FIGURE D.6: Temps d'exécution normalisé des benchmarks sur la plate-forme NUMA 32 avec deux tailles d'entrée de données.

	Par Défaut		**Best Tuning**	
	courte	**intermédiaire**	**courte**	**intermédiaire**
`bang`	1.48	1.32	1.10	1.13
`big`	0.92	0.93	0.92	0.93
`ehb`	1.27	1.14	1.27	1.12
`orbit_int`	2.50	1.35	2.08	1.23
`serialmsg`	1.04	1.00	1.00	1.00
`timer_wheel`	2.13	1.89	2.13	1.88

TABLE D.3: *Speedups* avec la configuration proposée sur la plate-forme NUMA 32

D.4.4.2 Applications Réelles

Après avoir évalué notre prototype avec les benchmarks de la suite BenchErl, nous avons procédé à l'évaluation des applications réelles. Pour cette évaluation, nous avons utilisé les applications Sim-Diasca et ErlangTW. La Figure D.7 montre le temps d'exécution normalisé de ces applications par rapport à la configuration par défaut, c'est-à-dire, en prenant comme référence le temps d'exécution en utilisant la VM sans

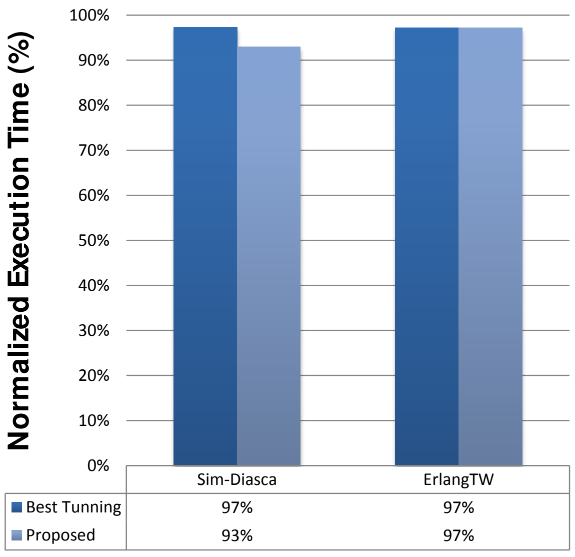


FIGURE D.7: Temps d'exécution normalisé de Sim-Diasca et ErlangTW sur la plate-forme NUMA 32

	Par Défaut	Best Tuning
Sim-Diasca	1.08	1.05
ErlangTW	1.03	1.00

TABLE D.4: Speedups mesurés pour Sim-Diasca et ErlangTW avec la configuration proposée sur la plate-forme NUMA 32.

paramètres optionnels. Pour Sim-Diasca nous avons réussi à attendre des améliorations de ∼8 % et ∼5 % en temps d'exécution lorsque l'on compare les configurations par défaut et best-tuning. Même si ces améliorations de performances ne sont pas aussi évidentes que celles que nous avons obtenues avec les benchmarks de BenchErl, elles sont encore importantes si l'on considère qu'elles ont été obtenues avec des modifications mineures dans le code de la simulation, c'est-à-dire, l'inclusion des *hints* pour effectuer le marquage de hubs. Pour Sim-Diasca les facteurs qui ont influencé la performance le plus ont été, par ordre d'importance, le vol de travail hiérarchique, l'équilibreur de charge hiérarchique et l'association d'ordonnanceurs aux PUs.

Nous n'avons pas réussi une amélioration de performance si importante avec ErlangTW. Nous l'avons pu améliorer dans seulement ∼3 %, à peu près la même amélioration que nous avons obtenue en utilisant seulement les options disponibles sur la VM originale. Le Tableau D.4 apporte la liste complète des speedups pour ces deux applications tenant compte des configurations par défaut et best-tuning comme référence. Le Chapitre 5 contient une explication détaillée des raisons de cette faible amélioration.

Les résultats obtenus par l'exécution de Sim-Diasca et ErlangTW nous montrent que, lorsque l'application correspond à nos hypothèses sur le domaine du problème, nous pouvons en effet créer un RE avec de meilleures performances sur les plates-formes NUMA.

D.4.5 Conclusions

Cette section a présenté notre prototype de RE pour les architectures NUMA. Nous avons modifié la machine virtuelle Erlang pour mettre en œuvre nos propositions et nous les avons évaluées en utilisant des applications et des benchmarks. En outre nos avons créé la possibilité pour le développeur de rajouter nouvelles stratégies d'équilibrage de charge, vol de travail et placement initial des acteurs. D'entre autres modifications, nous pourrons mettre en évidence la création de stratégies hiérarchiques d'équilibrage de charge et de vol de travail, les politiques *scatter* et *compact* pour le placement initial, la politique d'allocation retardée de mémoire pour les tas des acteurs et allocation locale de mémoire dans les plates-formes NUMA.

Pour évaluer l'efficacité de notre proposition, nous avons évalué la performance d'une version modifiée de la machine virtuelle Erlang. Nos tests montrent que si l'application cible est conforme nos suppositions du domaine du problème, les optimisations qui tiennent en compte les aspects NUMA de la plate-forme sous-jacente peuvent apporter des gains de performance importants.

D.5 Travaux Connexes

La recherche sur l'utilisation efficace des plates-formes NUMA est relativement commune. Cet intérêt peu être du à la croissante adoption de ces plates-formes et aussi à ses caractéristiques particulières d'accès à la mémoire. Même si ces architectures sont maintenant populaires, des articles publiés sur l'utilisation des ces machines par des REs basés sur le modèle d'acteur sont rares. Dans cette section nous décrivons quelques travaux qui nous considérons comme les plus relevants. Nous nous concentrons sur les travaux qui traitent les aspects à l'exécution que nous avons investigués dans cette thèse.

Nous avons divisé la description de ces travaux dans deux catégories. La première contient les travaux qui sont directement liés au modèle d'acteurs. La deuxième comprendre les travaux liés à l'utilisation efficace des plate-formes NUMA en général.

D.5.1 Approches Liées au Modèle d'Acteurs

Plusieurs *frameworks* pour la programmation basé sur le modèle d'acteurs existent. Ce ne serait pas possible les décrire tous ici. Pour cela, nous discutons quelques travaux sélectionnés. Ces travaux ont été sélectionnés par leur similarité avec l'approche que nous avons proposé. Même s'il existe plusieurs approches pour l'exécution efficace de REs, ces travaux se concentrent sur les plates-formes SMP.

La VM Erlang En utilisant certains des *benchmarks* que nous avons utilisés dans notre évaluation, Zhang [Zha11] caractérise la *scalability* de la VM Erlang sur le processeur multi-cœurs TilePro64. Comme une tentative de réduire la contention causée par des verrous dans le cadre de *ETS tables*[5], Nyblom [Nyb11] propose l'utilisation de mémoire transactionnelle logicielle pour le contrôle des accès à la structure de données commune avec des résultats encourageants.

Le compromis entre la performance et la la consommation de mémoire des tas privés, hybrides et partagés dans des langages concurrents est exploré par Carlsson *et al.* [CSW03]. Johansson *et al.* [JSW02] évaluent la performance de ces architectures de tas sur la VM Erlang.

Le projet RELEASE [BCC+12, 42] vise la création d'un paradigme de haut niveau pour le logiciel de serveur à grande échelle. La suite BenchErl de *benchmarks* [APR+12] que nous avons utilisé dans nos tests a été conçue par ce projet. RELEASE a abordé plusieurs aspects de la performance importants liés à la VM Erlang : compilation de code natif, optimisations concernant l'échange de messages, le profilage et l'exécution distribuée [PRL13, SST12, LT13, PS12].

Bien que l'ensemble de ces aspects peuvent être utilisés pour améliorer les performances de la VM Erlang sur des plates-formes NUMA, ils sont aussi génériques dans le sens où ils ne sont pas spécifiques pour cet architecture. Ils sont aussi importants sur les plates-formes NUMA que sur les plates-formes SMP.

[5]http://www.erlang.org/doc/man/ets.html

Autres Environnements d'Éxecution Dans cette section nous présentons quelques REs qui ont des caractéristiques similaires à notre travail. Nous commençons par l'introduction de deux REs : Kilim et Akka. En suite, nous concluons avec la discussion sur Charm++. Même si Charm++ n'est pas vraiment basé sur le modèle d'acteur, il partage quand-même la plupart de ses caractéristiques et possède quelques approches aux plates-formes NUMA semblables à la nôtre.

► **Kilim et Akka** [SM08, Gup12] Ces deux *frameworks* ont des mécanismes au niveau des applications qui permettent aux développeurs de lier l'exécution d'un groupe d'acteurs à un nœud spécifique d'une machine NUMA. Cela permet aux développeurs d'applications d'améliorer manuellement les performances du système dans ces plates-formes. Une utilisation possible de ces mécanismes serait la création manuelle d'un *scheduler* (Kilim)/*dispatcher* (Akka) pour chaque nœud NUMA. De cette façon, l'affinité des acteurs pourrait être facilement maintenue. D'autre part, l'équilibrage de charge doit être faite manuellement par le développeur de l'application. Une intervention manuel peut être suffisante pour une application et plate-forme matérielle [SM08] spécifiques, mais une telle solution manque de la portabilité. Une solution manuel plus complexe n'est pas, à notre avis, souhaitable car nous voulons que le développeur de l'application se concentre sur l'application et pas sur l'environnement d'exécution dans lequel son application sera exécutée.

► **Charm++** est un langage de programmation parallèle basé sur C++ [KK93]. Les applications fonctionnant sur cette plate-forme sont composés de plusieurs entités communicantes appelées *chares*. Les chares se communiquent par l'échange des messages asynchrones. Ils peuvent être migrés et, comme dans le modèle d'acteur, le RE est responsable d'assurer la transparence de localisation. Charm++ a un concept similaire à la politique de placement initial que nous avons défini. Dans Charm++ ce concept est appelé *seed load-balancers* [43].

L'ordonnancement de chares est orienté à messages, c'est-à-dire, il n'y a pas de préemption et le changement de contexte est coopérative. Dans un RE d'acteurs la famine est évitée à travers de la préemption ou de la création des *threads* supplémentaires. De plus, les acteurs sont généralement à grain plus fin : un acteur peut être créé pour effectuer une tâche très courte et spécifique. En outre, alors que l'état du système est normalement maintenu par des acteurs, ce n'est pas le cas typique dans les applications Charm++.

D.5.2 Approches Générales pour les Plates-Formes NUMA

Des chercheurs de la communauté de parallélisme ont démontré un vif intérêt pour solutions que ne sont pas basées sur le modèle d'acteurs. Les solutions qu'ils ont proposées comprennent des solutions de haut et bas niveau.

► **MPI** [For12] est un des outils prédominants dans le domaine des applications HPC. La norme MPI définit un ensemble de fonctions pour spécifier la topologie de la plate-forme sous-jacente. Ces fonctions peuvent être utilisées pour créer un communicateur MPI spécifique dans lequel les rangs de processus sont réorganisés pour une meilleure association entre les processus et les PUs. Rashti [M.J11] *et al.* montrent comment une meilleure adéquation entre les rangs de processus MPI et la topologie physique peut apporter des gains importants dans les performances de communication. Avec de bons résultats de performance dans une approche similaire [MCO09, JMT13, LWZ13] effectuent une réorganisation des rangs des processus en prenant en compte non seulement le réseau et les caractéristiques de la plate-forme NUMA numa mais aussi le graphe de communication de l'application.

► **Java** utilise des mécanismes de *gargabe collection* qui ont été modifiés par Ogasawara [Oga09] pour créer un gestionnaire de mémoire pour les architectures NUMA. Dans cette approche innovatrice, le gestionnaire de mémoire prend des décisions sur le placement des objets au cours des cycles du algorithme de récupérateur de mémoire. Pendant son opération, les objets sont marqués conformément aux accès qui ont été effectués. Parmi les *threads*, celui qui accède le plus un objet particulier est défini comme son *thread* dominant. Ensuite, l'algorithme modifié de récupération de mémoire copie cet objet à partir de son emplacement (nœud) actuel vers le nœud dans lequel le *thread* dominant se trouve. Cette solution est similaire à nos allocateurs de mémoire dans lequel la mémoire qui sera utilisée par un *thread* est allouée dans le même nœud où le *thread* se trouve.

► **OpenMP** [CCD+06] possède plusieurs outils pour assurer une exécution efficace sur les plates-formes NUMA. Nikolopoulos *et al.* [NPP+00] présentent un mécanisme de migration de pages basé sur un profilage initial des premières itérations d'une application OpenMP. Duran *et al.* [DPA+08] proposent quelques extensions de la norme OpenMP afin que l'affinité entre les *threads* et les données puisse être tracée. En utilisant BubbleSched [TNW07] pour le regroupement hiérarchique et le placement des *threads* et MaMI [41] pour effectuer des migrations de pages de mémoire, ForestGOMP [BFG+10, BAG+10] présente un ordonnanceur de *threads* multi-niveaux combiné avec un gestionnaire NUMA de mémoire. L'ordonnancement des tâches OpenMP sur une plate-forme NUMA est fait par Broquedis *et al.* [BFG+09] et par Olivier *et al.* [OPW+12]. Cependant, le modèle de tâches présenté par OpenMP est différent de celui du modèle d'acteur dans quelques aspects très importants tels que la granularité des tâches et de la communication.

► **Systèmes d'exploitation** Des solutions au niveau du noyau du système d'exploitation comme AutoNuma et NumaSched [SS12, 44, 45] montrent que des améliorations dans les environnements d'exécution ne sont pas les seules possibilités. Ces approches tentent d'améliorer les performances de manière transparente en faisant une meilleure allocation de la mémoire et un meilleur ordonnancement des processus. En particulier, NumaSched a aussi la notion d'un home node par processus. Un processus aura la mémoire qu'il utilise allouée préférentiellement dans son home node. L'ordonnanceur ira restreindre l'exécution d'un processus à son home node sauf si l'équilibrage de charge en décide autrement. Dans ce cas, une migration peut finir par la modification du home node du processus. Cette étape sera suivie par une migration de page de mémoire de une manière paresseuse. NumaSched, d'autre part, utilise une approche différente. Pour chaque processus, le noyau maintient les derniers nœuds NUMA des pages de mémoire qui ont été accédés. De même, pour chaque page de mémoire, le noyau maintient le dernier nœud NUMA qui lui a accédé. Sur la base de ces statistiques, le noyau décide si (et où) un processus ou une page de mémoire doit être migrée. Malheureusement, ce type d'approche a une efficacité limitée sur les environnements d'exécution qui n'ont pas un lien direct entre chaque flux interne d'exécution et un *thread*. En outre, l'environnement d'exécution a des informations supplémentaires, qui ne sont pas disponibles au noyau, qui peuvent être utilisées pour effectuer un ordonnancement et une allocation de mémoire plus efficaces.

Malgré l'existence de plusieurs solutions pour l'utilisation efficace des machines NUMA, nous considérons que notre proposition d'utiliser le modèle d'acteur dans ces plates-formes hiérarchiques soit essentielle. Contrairement à la plupart des solutions alternatives, le modèle d'acteur offre une interface de programmation de haut niveau et permet au développeur écrire des applications qui sont totalement découplées de l'architecture matérielle sous-jacente. En outre, comme la disponibilité de plates-formes matérielles avec un grand nombre de cœurs et processeurs continue d'augmenter, nous pouvons espérer une croissance de la demande pour solutions parallèles avec des bonnes performances et qui soient faciles à utiliser.

D.6 Conclusion

Les fabricants de *hardware* s'affrontent dans une course sans fin pour produire des processeurs de plus en plus performants. Afin de remédier à la complexité croissante pour le développement des nouveaux processeurs et aussi pour éviter la haute consommation énergétique, une de leurs dernières stratégies dans cette course est justement la création de processeurs avec un nombre toujours croissant de cœurs. En effet, cette stratégie a renversé la charge de la parallélisation de concepteurs de matériel vers les développeurs de logiciels.

Les outils de programmation parallèles existent depuis longtemps, mais ils ont été principalement utilisés par les développeurs de logiciel de calcul scientifique. Mais la majorité de développement de logiciel a complètement ignoré les aspects de concurrence et de programmation parallèle [Sut05]. C'est seulement après l'arrivée des processeurs multi-cœurs dans le marché grand public que nous commençons à voir une augmentation générale d'intérêt pour ces outils. Dernièrement, le modèle d'acteur, qui avait été créé au début des années soixante-dix [HBS73] (milieu des années quatre-vingt, si nous le considérons dans la forme que nous tenons aujourd'hui [Agh86]), a enregistré un taux sans précédent d'adoption en raison non seulement de ses caractéristiques intrinsèques (absence de l'état partagé et donc pas de verrous ou de synchronisation), mais aussi à l'émergence des RE basés sur le modèle d'acteur de haute performance telles que la VM Erlang.

Comme l'adoption du modèle d'acteur a grandi, ces applications ont commencé peu à peu à faire leur apparition sur les plates-formes plus performantes. Ces plates-formes possèdent souvent une mémoire hiérarchique (NUMA) et utilisent des processeurs multi-cœurs. Nous avons décrit dans ce texte comment les REs pour les acteurs sont fréquemment optimisés pour les machines SMP. D'un autre coté, le même constat ne peut pas être affirmé à propos de l'architecture NUMA. Cela est vrai même si en réalité plusieurs systèmes essentiels sont basés sur le modèle d'acteurs et tournent sur des architectures NUMA.

Dans cette thèse, nous analysons certaines de ces applications d'acteurs à la recherche de caractéristiques communes. Avec une connaissance plus approfondie sur les applications et le comportement de la plate-forme matérielle sous-jacente, nous proposons, réalisons et évaluons une série de propositions utilisant un RE réel basé sur le modèle d'acteur, la VM Erlang. Ces propositions tiennent compte de l'application et des particularités du matériel pour faire un meilleur ordonnancement, équilibrage de charge et gestion de la mémoire. Ces modifications ont été apportées au niveau de le RE, donc aucune modification sur les applications elles-mêmes est nécessaire. Cependant, le développeur de l'application peut également fournir au RE des indications sur le comportement de son application, en l'aidant ainsi à prendre de meilleures décisions et d'améliorer par conséquent, les résultats de notre approche.

D.6.1 Contributions

Après une brève introduction (Section D.1) et la présentation des concepts de base, nous présentons la première contribution de cette thèse. L'analyse et la caractérisation des applications d'acteurs (Section D.2). D'abord nous analysons les caractéristiques générales des acteurs tels que la durée de vie, la taille des messages et les coûts de communication. Ensuite, nous observons les propriétés d'exécution spécifiques des applications, spécialement les interactions des acteurs. Au cours de cette observation nous nous rendons compte que certains acteurs sont beaucoup plus connectés que d'autres. Ces acteurs sont responsables de la création de la majorité des autres acteurs et sont impliqués dans la plupart des communications. Nous appelons ces acteurs particuliers *hubs*. Notre analyse montre aussi que les acteurs ont tendance à communiquer seulement avec un petit sous-ensemble d'acteurs. Ce petit sous-ensemble est généralement créé par un hub. Nous définissons l'ensemble des acteurs avec lesquels un acteur communique comme son groupe d'affinité.

Dans la Section D.3 nous prenons les résultats de notre précédente contribution et la connaissance des architectures NUMA en considération pour proposer une approche hiérarchique concrète à la création d'un RE efficace. Nous discutons des politiques de placement initial et le concept de distances de l'ordonnanceur, donnant au développeur de l'application la liberté de choisir la fonction pour calculer la distance qui est la plus approprié à son contexte. Les valeurs fournies par cette fonction sont au cœur de notre approche hiérarchique pour les migrations d'acteurs. Ensuite, nous présentons notre équilibreur de charge hiérarchique et des algorithmes de vol de travail. Ces algorithmes de migration tentent de garder les acteurs près du nœud dans lequel leur mémoire est allouée, c'est-à-dire, son *home node*. Dans ce but, les algorithmes de migration proposés non seulement évitent migrations qui les éloignent de leurs home nodes mais aussi ces algorithmes essaient de les ramener près de leurs home nodes. Lorsque la migration entre les nœuds NUMA est inévitable, les algorithmes tentent de minimiser la distance entre l'acteur et son home node. Pour cela, ils utilisent le vecteur de distances pré-calculé et gardé par chaque ordonnanceur.

Nous avons appliqué et évalué la performance de notre proposition sur la VM d'Erlang. Pour ce faire, nous avons évalué la performance de la VM modifiée en utilisant à la fois des benchmarks standards et des applications réelles. Nos expériences montrent avec succès que, quand on considère la performance offerte par la VM Erlang standard, certaines politiques simples d'optimisation qui prennent compte l'architecture hierarchique de la machine peuvent apporter des améliorations de performances significatives (jusqu'à un facteur de 2,50). Nous avons également montré que les applications mal codées, qui pourraient afficher des performances acceptables sur des architectures SMP, peuvent subir des pertes de performance graves sur ces plates-formes de mémoire partagée hiérarchique.

En plus des nouvelles stratégies de vol de travail, d'équilibrage de charge et de placement initial des acteurs, nous avons ajouté de nouvelles fonctionnalités à la VM Erlang telles que l'allocation dans la mémoire locale et l'allocation retardée des tas. Pour cela nous avons utilisé le patron de conception Stratégie, ce qui signifie que de nouvelles politiques peuvent être facilement créées. Afin de mettre en œuvre nos modifications, nous avons également développé quelques outils satellites pour le traçage et la visualisation de graphes de communication (ces outils ont été utilisés pour générer la Figure 3.5) et pour effectuer l'association définitive d'acteurs aux PUs. Nos avons aussi développé d'autres outils comme une outil pour accéder aux compteurs matériels de performance (utilisé pour générer la Figure 3.4), et Jhwloc une interface Java pour utiliser hwloc.

La plupart des contributions décrites ci-dessus ont été publiées dans des événements internationaux. L'Appendice A contient la liste complète des documents publiés au cours de l'élaboration de cette thèse.

D.6.2 Perspectives

Même si l'approche que nous proposons est déjà suffisante pour apporter des améliorations de performances significatives pour les REs, elle nous conduit aussi à quelques perspectives de recherche. D'abord nous avons l'intention de rendre ces nouvelles fonctionnalités disponibles pour être introduites dans la VM Erlang officielle. Actuellement, la VM modifiée est disponible comme une branche du dépôt Git original de la VM Erlang OTP. Nous avons l'intention d'optimiser notre code et de l'adapter aux normes de la distribution Erlang OTP afin que nous puissions soumettre un patch pour l'inclusion. Ces changements doivent être appliqués à d'autres systèmes d'exploitation comme FreeBSD et d'améliorer le support pour différentes architectures matérielles. Nous envisageons aussi l'intégration de notre approche dans d'autres REs basés sur le modèle d'acteur tels que Kilim et Akka. En outre, une comparaison des performances plus profonde entre les REs de haut niveau, tels qu'Erlang et Akka, et ceux qui utilisent des approches plus légères, tels que Charm++, pourrait préciser les coûts généraux encourus par le choix d'un environnement de plus haut niveau.

Dans ce travail, nous avons abordé plusieurs aspects de l'exécution d'une application base sur le modèle d'acteur. En particulier, nous avons analysé et proposé une nouvelle approche pour traiter les hubs et les groupes d'affinité. Nous avons discuté de la performance et de la communication des acteurs et nous avons montré comment ces aspects peuvent être pris en considération pour diminuer la surcharge imposée par architectures hiérarchiques à mémoire partagée. Nous présentons maintenant quelques perspectives de recherche possibles sur ces sujets.

► **Hubs et Groupes d'Affinité -** L'identification des hubs dans notre approche actuelle dépend de *hints* fournis par le développeur de l'application. Dans ce travail, nous avons analysé les gains de performance d'une identification correcte et légère peut avoir sur une application. Un mécanisme dynamique de détection et de placement de hubs rendrait notre approche encore plus transparente pour l'utilisateur. Il y a, cependant, de nombreux défis à la détection automatique des hubs. Ces difficultés sont principalement liées à la taille et à la nature dynamique des graphes de communication des acteurs. En outre, après le placement initial dés hubs, aucun rééquilibrage de charge est effectué. Par conséquent, actuellement, les hubs ont leur exécution restreinte à l'ordonnanceur choisi par la politique de placement initial.

Nous supposons que les hubs sont, pour la plupart du temps, responsables de la création de la majorité des autres acteurs qui appartiennent à son groupe d'affinité. Ceci est basé sur les différentes heuristiques de conception proposées dans la Section D.3. Toutefois, ces heuristiques ne peuvent pas être efficaces pour toutes les applications. Notre approche utilise la position actuelle et le home node comme une indication de la proximité de l'acteur à son hub et, par conséquent, son groupe d'affinité. En éliminant la nécessité d'utiliser cette heuristique, nous nous attendons à améliorer l'efficacité de notre approche. Pour cela, nous envisageons d'employer des techniques de partitionnement du graphe de communication en utilisant des outils tels que Scotch [46] dans une approche similaire à celle de Jeannot *et al.* [JMT13] et Li *et al.* [LWZ13].

► **Communication sur les Plates-formes Hiérarchiques -** Les groupes d'affinité indiquent quels acteurs sont plus susceptibles de communiquer entre eux. La performance de la communication dépend de la proximité des acteurs avec leur groupe d'affinité, mais aussi de l'organisation de la mémoire choisie par le RE. La façon dont le tas des acteurs est organisé et leur impact sur la performance du RE ont été étudiés à fond sur les plates-formes SMP [CSW03, JSW02]. Toutefois, des recherches sur l'impact des différents organisations de tas sur les REs tournant sur des plates-formes NUMA est nécessaire. Même si un tas de mémoire partagée global n'est pas la meilleure option pour une machine NUMA, les échanges de messages pouvaient être encore améliorées par la mise en œuvre d'une architecture de tas basée sur mémoire partagée pour les acteurs à l'intérieur du même groupe d'affinité, ou à l'intérieur du même nœud NUMA, dans lequel les messages seraient passés par valeur. En outre, comme optimisation pour l'exécution, la surcharge provoquée par l'utilisation de verrous pour accéder aux boîtes aux lettres des acteurs pourrait éventuellement être diminuée avec l'utilisation d'autres techniques telles que la mémoire transactionnelle [DDS+10, HLR10] ou les structures de données sans verrou.

Les échanges de messages ne sont pas, pourtant, les seuls coûts de communication qui doivent être payés par ces applications sur les plates-formes hiérarchiques. Les performances d'E/S sur les plates-formes NUMA sont également variables en fonction du nœud NUMA où le processus exécute. Chaque dispositif d'E/S comme, par exemple, un disque dur ou une carte réseau, est relié à un nœud NUMA spécifique. Un acteur qui effectue des opérations d'E/S sur ces machines devrait être programmé pour fonctionner spécifiquement sur ces nœuds pour éviter le trafic sur l'interconnexion NUMA et donc améliorer les performances. En outre, des recherches sur les raisons de la faible performance présentée par la plate-forme Altix UV 2000 (Appendice C) est clairement nécessaire.

La communication entre les cœurs sur un processeur multi-cœurs est effectuée grâce à l'utilisation d'un NoC. Souvent, les interconnexions du NoC ne sont pas un graphe complet, donc les performances de communication entre les cœurs peuvent dépendre de la distance topologique entre eux. Par exemple, la famille de puces Tile-Gx de Tilera emploie une interconnexion en grille entre les cœurs [47]. D'autre part, le MPPA-256 manycore de Kalray [ABB+13, DML+13] emploie une topologie de tore 2D. Les interconnexions du NoC ressemblent à celles d'une plate-forme NUMA et, pour cette raison, nous avons l'intention d'approfondir nos recherches sur l'adaptation de notre approche aux plates-formes manycore tels que MPPA-256 [FGM13b].

Bibliography

[ABB$^+$13] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Dore, Paul Dubrulle, Benoît Dupont de Dinechin, et al. Extended cyclostatic dataflow program compilation and execution for an integrated many-core processor. *Procedia Computer Science*, 18:1624–1633, 2013.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[ALS10] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide*. O'Reilly, 2010.

[AMST97] Gul Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[APR$^+$12] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. A scalability benchmark suite for erlang/otp. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang*, Erlang '12, pages 33–42. ACM, 2012.

[Arm07] Joe Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.

[Arm10] Joe Armstrong. Erlang. *Communications of the ACM*, 53:68–75, September 2010.

[AV90] J.L. Armstrong and S.R. Virding. Erlang - an experimental telephony programming language. In *Switching Symposium, 1990. XIII International*, volume 3, pages 43 –48, may-1 jun 1990.

[BA99] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[BAG+10] François Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, P-A Wacrenier, and Raymond Namyst. Structuring the execution of openmp applications for multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[BB03] Albert-László Barabási and Eric Bonabeau. Scale-free networks. Scientific American, May 2003.

[BBH+04] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T Marr, J Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and KS Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1), 2004.

[BBS+00] David M Brooks, Pradip Bose, Stanley E Schuster, Hans Jacobson, Prabhakar N Kudva, Alper Buyuktosunoglu, J Wellman, Victor Zyuban, Manish Gupta, and Peter W Cook. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *Micro, IEEE*, 20(6):26 –44, nov/dec 2000.

[BCC+12] O. Boudeville, F. Cesarini, N. Chechina, K. Lundin, N. Papaspyrou, K. Sagonas, S. Thompson, P. Trinder, and U. Wiger. RELEASE: a high-level paradigm for reliable large-scale server software. In *Proceedings of the Symposium on Trends in Functional Programming*, 2012.

[BCOM+10] François Broquedis, Jérôme Clet Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa Italie, 02 2010.

[BFG+09] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic task and data placement over numa architectures: an openmp runtime perspective. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 79–92. Springer, 2009.

[BFG+10] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal on Parallel Programming, Special Issue on OpenMP; Guest Editors: Matthias S. Müller and Eduard Ayguade*, 38(5):418–439, 2010.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216. ACM, 1995.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices*, 35(11):117–128, November 2000.

[Bou13] Olivier Boudeville. Some Information About The Sim-Diasca City-Example Benchmarking Case. Technical report, EDF, March 2013.

[BR04] Béla Bollobás and Oliver Riordan. Robustness and vulnerability of scale-free random graphs. *Internet Mathematics*, 1(1):1–35, 2004.

[But97] David R Butenhof. Posix threads programming. *Parallel Computing*, pages 1–33, 1997.

[CCD+06] Rohit Chandra, Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2006.

[CFNM13] Márcio Castro, Emilio Francesquini, Thomas M. Nguélé, and Jean-François Méhaut. Analysis of Computing and Energy Performance of Multicore, NUMA, and Manycore Platforms for an Irregular Application. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, IA^3'13, pages 5:1–5:8. ACM, 2013.

[CPRA$^+$12a] E. Cruz, Christiane Pousa Ribeiro, M. Alves, Alexandre Carissimi, Philippe O. A. Navaux, and Jean-François Mehaut. Memory-aware Thread and Data Mapping for Hierarchical Multi-core Platforms. *International Journal on Networking and Computing*, 2(1):96–116, 2012.

[CPRA$^+$12b] E. Cruz, Christiane Pousa Ribeiro, M. Alves, Alexandre Carissimi, Philippe O. A. Navaux, and Jean-François Mehaut. Using Memory Access Traces to Map Threads on Hierarchical Multi-core Platforms. *Intl. Journal on Networking and Computing*, 2(1):96–116, 2012.

[CST$^+$10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154. ACM, 2010.

[CSW03] Richard Carlsson, Konstantinos Sagonas, and Jesper Wilhelmsson. Message analysis for concurrent languages. In Radhia Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 73–90. Springer Berlin Heidelberg, 2003.

[CT09] Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition edition, 2009.

[DDS$^+$10] Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, and Michael Spear. Transactional mutex locks. In *Euro-Par 2010-Parallel Processing*, pages 2–13. Springer, 2010.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[DML$^+$13] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[DPA+08] Alejandro Duran, Josep M Perez, Eduard Ayguadé, Rosa M Badia, and Jesus Labarta. Extending the openmp tasking model to allow dependent tasks. In *OpenMP in a New Era of Parallelism*, pages 111–122. Springer, 2008.

[Eva06] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. *Proceedings of BSDCan*, 2006.

[FGM12] Emilio Francesquini, Alfredo Goldman, and Jean-François Mehaut. Towards Automatic Actor Pinning on Multi-core Architectures. In *ACM SIGPLAN Erlang Workshop, Erlang'12*, Copenhagen, Denmark, sep 2012.

[FGM13a] Emilio Francesquini, Alfredo Goldman, and Jean-François Mehaut. Actor Scheduling for Multicore Hierarchical Memory Platforms. In *Proceedings of the 12th ACM Erlang Workshop*, Boston, US, sep 2013. ACM SIGPLAN.

[FGM13b] Emilio Francesquini, Alfredo Goldman, and Jean-François Méhaut. Improving the performance of actor model runtime environments on multicore and manycore platforms. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '13, pages 109–114. ACM, 2013.

[For12] MPI Forum. MPI: A Message-Passing Interface Standard, September 2012.

[GM11] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[Gup12] Munish Gupta. *Akka Essentials*. Packt Publishing Ltd, 2012.

[Hal09] S. Halloway. *Programming Clojure*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2009.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[Hed98] Pekka Hedqvist. A parallel and multithreaded ERLANG implementation. Master's thesis, Computer Science Department, Uppsala University, Uppsala, Sweden, 1998.

[HLR10] Tim Harris, Jim Larus, and Ravi Rajwar. Transactional memory (synthesis lectures on computer architecture). *Synthesis Lectures on Computer Architecture. Morgan and Claypool*, 2010.

[HO07] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Coordination Models and Languages*, pages 171–190. Springer, 2007.

[HO09] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202 – 220, 2009. Distributed Computing Techniques.

[IKBW$^+$79] Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, John G. P. Barnes, Olivier Roubine, and Jean-Claude Heliard. Rationale for the design of the ada programming language. *SIGPLAN Not.*, 14(6b):1–261, June 1979.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, July 1985.

[JHVG95] Ralph Johnson, Richard Helm, John Vlissides, and Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[JMT13] E. Jeannot, G. Mercier, and F. Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. volume PP, pages 1–1, 2013.

[Jon86] Douglas W Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, 1986.

[JSW02] Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson. Heap architectures for concurrent languages using message passing. In *Pro-*

ceedings of the 3rd International Symposium on Memory Management, ISMM '02, pages 88–99. ACM, 2002.

[Kal13] Laxmikant Kale. The Coming Era of Adaptive Control Systems in HPC, October 2013. Keynote Speech at the 42nd International Conference on Parallel Processing, ICPP 2013.

[Kam09] Patryk Kaminski. Numa aware heap memory manager. *AMD Developer Central*, 2009.

[KCS04] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[KK93] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 91–108. ACM, 1993.

[Kle05] Andi Kleen. A NUMA API for linux. Technical Report 4621437, Novell, 2005.

[KSA09] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: A comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, 2009.

[LADW05] Lun Li, David Alderson, John C. Doyle, and Walter Willinger. Towards a theory of scale-free graphs: Definition, properties, and implications. *Internet Mathematics*, 2:4, 2005.

[Lar09] James Larus. Spending Moore's dividend. *Communications of the ACM*, 52:62–69, May 2009.

[LT13] Huiqing Li and Simon Thompson. Multicore profiling for erlang programs using percept2. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 33–42. ACM, 2013.

[Lun08] Kenneth Lundin. Inside the Erlang VM with focus on SMP, November 2008.

[LWZ13] Dongyang Li, Yunlan Wang, and Wei Zhu. Topology-aware process mapping on clusters featuring numa and hierarchical network. In *Parallel and Distributed Computing (ISPDC), 2013 IEEE 12th International Symposium on*, pages 74–81. IEEE, 2013.

[McC95] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[MCO09] Guillaume Mercier and Jérôme Clet-Ortega. Towards an efficient process placement policy for mpi applications in multicore environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 104–115. Springer, 2009.

[MG12] Zoltan Majo and Thomas R. Gross. Matching memory access patterns and data placement for NUMA systems. In *Proceedings of the Tenth Intl. Symposium on Code Generation and Optimization*, CGO '12, pages 230–241. ACM, 2012.

[M.J11] M.J. Rashti *et al.* Multi-core and network aware MPI topology functions. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 50–60, 2011.

[MS$^+$96] Larry W McVoy, Carl Staelin, et al. LMbench: Portable Tools for Performance Analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[MS11] R. Mikkilineni and I. Seyler. Parallax - a new operating system for scalable, distributed, and parallel computing. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 976 –983, may 2011.

[MT99] Ian A. Mason and Carolyn L. Talcott. Actor languages their syntax, semantics, translation, and equivalence. *Theor. Comput. Sci.*, 220(2):409–467, 1999.

[NPP$^+$00] Dimitrios S Nikolopoulos, Theodore S Papatheodorou, Constantine D Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, pages 95–103. IEEE, 2000.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[NVI13] NVIDIA. Tegra 4 Family, February 2013.

[Nyb11] Patrik Nyblom. Erlang ets tables and software transactional memory: how transactions make ets tables more like ordinary actors. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Erlang '11, pages 2–13, 2011.

[OAC$^+$04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stééphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[Oga09] Takeshi Ogasawara. Numa-aware memory manager with dominant-thread-based copying gc. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 377–390. ACM, 2009.

[OPW$^+$12] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *International Journal of High Performance Computing Applications*, 26(2):110–124, 2012.

[Pak07] Scott Pakin. The design and implementation of a domain-specific language for network performance testing. *Parallel and Distributed Systems, IEEE Transactions on*, 18(10):1436–1449, 2007.

[PJS12] Martin Fowler Pramodjumar J Sadalage. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[PRC$^+$12] Laercio Lima Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, Chao Mei, Abhinav Bhatele, Philippe O.A. Navaux, Francois Broquedis, Jean-Francois Mehaut, and Laxmikant V. Kale. A hierarchical approach for load balancing on parallel multi-core systems. *Proceedings of the 41st International Conference on Parallel Processing*, 0:118–127, 2012.

[PRC$^+$14] Laércio L. Pilla, Christiane P. Ribeiro, Pierre Coucheney, François Broquedis, Bruno Gaujal, Philippe O.A. Navaux, and Jean-François Méhaut. A topology-aware load balancing algorithm for clustered hierarchical multi-core machines. *Future Generation Computer Systems*, 30(0):191 – 201, 2014. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, ICPADS 2012 Selected Papers.

[PRL13] S. Thompson P. Rodgers, R. Baker and H. Li. Multi-level visualization of concurrent and distributed computation in erlang. In *Visual Languages and Computing (VLC) in The 19th International Conference on Distributed Multimedia Systems (DMS 2013)*, August 2013.

[PS12] Nikolaos Papaspyrou and Konstantinos Sagonas. On preserving term sharing in the erlang virtual machine. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, Erlang ’12, pages 11–20. ACM, 2012.

[Qui04] Michael Jay Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, NY, 2004.

[RMC+09] Christiane Pousa Ribeiro, J-F Mehaut, Alexandre Carissimi, Marcio Castro, and Luiz Gustavo Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on*, pages 59–66. IEEE, 2009.

[SJ11] Marjan Sirjani and MohammadMahdi Jaghoori. Ten years of analyzing actors: Rebeca experience. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 20–56. Springer Berlin Heidelberg, 2011.

[SKZ+11] T. Song, D. Kaleshi, Ran Zhou, O. Boudeville, Jing-Xuan Ma, A. Pelletier, and I. Haddadi. Performance evaluation of integrated smart energy solutions through large-scale simulations. In *Smart Grid Communications (SmartGridComm), 2011 IEEE International Conference on*, pages 37–42, 2011.

[SLVW08] Martin Sulzmann, Edmund SL Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In *Coordination Models and Languages*, pages 315–330. Springer, 2008.

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 104–128. Springer-Verlag, 2008.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[Sri10] Sriram Srinivasan. Kilim: A server framework with lightweight actors, isolation types and zero-copy messaging. Technical Report UCAM-CL-TR-769, University of Cambridge, Computer Laboratory, 2010.

[SS12] Prerna Saxena and Vaidyanathan Srinivasan. Optimizing virtual machine resource placement on multi-socket platforms. In *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*, pages 1–6. IEEE, 2012.

[SSR08] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48. ACM, 2008.

[SST12] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. Erllvm: An llvm backend for erlang. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, Erlang '12, pages 21–32. ACM, 2012.

[Ste02] Erik Stenman. *Efficient implementation of concurrent programming languages*. PhD thesis, Uppsala University, 2002.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[Tal00] Carolyn L. Talcott. Towards a toolkit for actor system specification. In Teodor Rus, editor, *AMAST*, volume 1816 of *Lecture Notes in Computer Science*, pages 391–406. Springer, 2000.

[TASS07] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared L2 caches on multicore systems in software. In *WIOSCA'07*, 2007.

[TDJ13] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 302–326. Springer-Verlag, 2013.

[TDM12] Luca Toscano, Gabriele D'Angelo, and Moreno Marzolla. Parallel discrete event simulation with erlang. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '12, pages 83–92. ACM, 2012.

[TKL$^+$12] Samira Tasharofi, RajeshK. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer Berlin Heidelberg, 2012.

[TNW07] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building portable thread schedulers for hierarchical multiprocessors: The bubblesched framework. In *Euro-Par 2007 Parallel Processing*, pages 42–51. Springer, 2007.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001. http://www.cs.rpi.edu/˜cvarela/oopsla2001.pdf.

[Vin11] Steve Vinoski. Yaws: Yet another web server. *Internet Computing, IEEE*, 15(4):90–94, 2011.

[Vin12] S. Vinoski. The Nitrogen Erlang Web Framework. *Internet Computing, IEEE*, 16(6):87–90, 2012.

[Zha11] Jianrong Zhang. Characterizing the scalability of erlang vm on many-core processors. Master's thesis, KTH, School of Information and Communication Technology (ICT), 2011. Trita-ICT-EX.

Online Resources

[1] A. Binstock. Interview with Donald Knuth, informIT.
http://www.informit.com/articles/article.aspx?p=1193856,
December 2008.

[2] Samsung. Press Release.
http://www.samsung.com/global/business/semiconductor/news-events/press-releases/detail?newsId=12521, 2013.

[3] Apple. Grand Central Dispatch (GCD) Reference, Mac OS X Reference Library.
https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html, July 2010.

[4] BSD. Grand Central Dispatch - FreeBSD port.
https://wiki.freebsd.org/GCD, December 2013.

[5] WhatsApp Blog. 1 million is so 2011.
http://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011/, January 2012.

[6] Bruno Capelas. Whatsapp chega a 430 milhões de usuários.
http://blogs.estadao.com.br/link/whatsapp-chega-a-430-milhoes-de-usuarios/, January 2014.

[7] Eugene Letuchy. Facebook Chat.
https://www.facebook.com/note.php?note_id=14218138919, 2008.

[8] The Associated Press. Number of active users at facebook over the years.

http://finance.yahoo.com/news/number-active-users-facebook-over-years-214600186--finance.html, October 2012.

[9] Alex Payne. Why scala? http://www.web2expo.com/webexsf2009/public/schedule/detail/6110, January 2009. Web 2.0 Expo, San Francisco.

[10] Intel Atom Processor. http://www.intel.com/content/www/us/en/processors/atom/atom-processor.html, 2013.

[11] Mac Developer Library. Thread Affinity API Release Notes for OS X. https://developer.apple.com/library/mac/releasenotes/Performance/RN-AffinityAPI/index.html, 2007. Version 10.5.

[12] Implementation and Ports of Erlang. http://www.erlang.org/faq/implementations.html.

[13] Akka Documentation. Dispatchers. http://doc.akka.io/docs/akka/snapshot/scala/dispatchers.html, 2013. Version 2.3 Snapshot.

[14] Erlang OTP Release Erlang 5.2/OTP R9B Highlights. http://www.erlang.org/documentation/doc-5.3/doc/highlights.html.

[15] The Linux man-pages project. Linux Programmer's Manual. https://www.kernel.org/doc/man-pages/, December 2013.

[16] John Levon and Philippe Elie. Oprofile: A system profiler for Linux. http://oprofile.sf.net, 2013.

[17] Glenn Kelman. Engineer-to-Engineer Talk: How and Why Twitter Uses Scala. http://blog.redfin.com/devblog/2010/05/how_and_why_twitter_uses_scala.html, May 2010.

[18] Apache Foundation. CouchDB NoSQL Database. http://couchdb.apache.org/, November 2013.

[19] Basho Technologies. Riak Distributed Database.
http://basho.com/riak/, November 2013.

[20] Jordan Novet. Basho Technologies takes aim at more enterprises with upgrades.
http://gigaom.com/2013/02/21/basho-technologies-takes-aim-at-more-enterprises-with-upgrades/, February 2013.

[21] Basho Technologies. Riak Users.
http://basho.com/riak-users/, December 2013.

[22] Amazon. SimpleDB Homepage.
http://aws.amazon.com/simpledb/, December 2013.

[23] Ali Ghodsi and Joe Armstrong. Apache vs. Yaws.
http://www.sics.se/~joe/apachevsyaws.html, 2007.

[24] Seth Falcon. The Making of Erchef, the Chef 11 Server.
http://www.getchef.com/blog/2013/02/15/the-making-of-erchef-the-chef-11-server/, 2013.

[25] Opscode. Facebook Likes Opscode and Private Chef.
http://www.getchef.com/press-releases/facebook-likes-opscode-and-private-chef/, February 2013.

[26] EDF. Sim-Diasca - Simulation of Discrete Systems of All Scales.
http://sim-diasca.org/, December 2013.

[27] Yanmin Zhang. Hackbench.
http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, 2008.

[28] OTP Design Principles User's Guide. Supervisor behaviour.
http://www.erlang.org/doc/design_principles/sup_princ.html, 2013. Version 5.10.4.

[29] Akka Documentation. Supervision and monitoring.

http://doc.akka.io/docs/akka/snapshot/general/supervision.html, 2013. Version 2.3 Snapshot.

[30] Erlang Efficiency Guide User's Guide. Profiling. http://www.erlang.org/doc/efficiency_guide/profiling.html, 2013. Version 5.10.4.

[31] Leah Rosin. Oracle Linux DTrace raises excitement, wariness. http://searchoracle.techtarget.com/feature/Oracle-Linux-DTrace-raises-excitement-wariness, February 2013.

[32] Frank C EIgler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. Architecture of systemtap: a linux trace/probe tool. http://sourceware.org/systemtap/archpaper.pdf, 2005.

[33] Björn Gustavsson. Wings 3D. http://www.wings3d.com/, December 2013.

[34] Evan Miller. Chicago Boss: A Rough Introduction. http://www.chicagoboss.org/tutorial.pdf, February 2012.

[35] Namdak Tonpa. N2O: No Bullshit Sane Framework for Wild Web. http://synrc.com/framework/web/n2o/doc/book.pdf, November 2013.

[36] HieSchella. Hierarchical scheduling for large scale architectures. http://forge.imag.fr/projects/hieschella/, 2012.

[37] Erlang Run-Time System Application (ERTS) Reference Manual. Erlang runtime system internal memory allocator library. http://erlang.org/doc/man/erts_alloc.html, 2013. Version 5.10.4.

[38] Doug Lea. A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html, 2000.

[39] GNU. The GNU C Library (glibc). http://www.gnu.org/software/libc/, 2013.

[40] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2009.

[41] Runtime Team. MaMI - Marcel Memory Interface. http://runtime.bordeaux.inria.fr/MaMI/, 2013.

[42] Release Project. A high-level paradigm for reliable large-scale server software. http://www.release-project.eu/, 2013.

[43] The Charm++ Parallel Programming System Manual. http://charm.cs.illinois.edu/manuals/html/charm++/, 2013. Version 6.5.0.

[44] A. Arcangeli. Linux kernel mailing list. [rfc] autonuma alpha10. https://lkml.org/lkml/2012/3/26/398, 2012.

[45] P. Zjilstra. Linux kernel mailing. on numa interfaces and stuff. http://lkml.org/lkml/2011/11/17/204, 2011.

[46] Scoth Project. http://scotch.gforge.inria.fr/, 2012.

[47] Tilera. Tilera Homepage. http://www.tilera.com/products/processors/TILE-Gx_Family, March 2013.

[48] The Linux man-pages project. Linux Administrator's Manual, NUMACTL(8). http://man7.org/linux/man-pages/man8/numactl.8.html, March 2004.