A Grid-Hypergraph Load Balancing Approach for Agent Based Applications in HPC Systems

Claudio Daniel Márquez Pérez

Escola d'Enginyeria
Departament d'Arquitectura de
Computadors i Sistemes Operatius

A Grid-Hypergraph Load Balancing Approach for Agent Based Applications in HPC Systems

Thesis submitted by Claudio Daniel Márquez Pérez in fullfillment of the requeriments for the degree of Doctor per la Universitat Autònoma de Barcelona, Ph.D. in High Performance Computing. This work has been developed in the Computer Architecture and Operating System Department of the Autonomous University of Barcelona under the advise of Dr. Eduardo César Galobardes and Dr. Joan Sorribes Gomis.

Bellaterra, June 13, 2017

A Grid-Hypergraph Load Balancing Approach for Agent Based Applications in HPC Systems

Thesis submitted by Claudio Daniel Márquez Pérez in fullfillment of the requeriments for the degree of Doctor per la Universitat Autònoma de Barcelona, Ph.D. in High Performance Computing. This work has been developed in the Computer Architecture and Operating System Department of the Autonomous University of Barcelona under the advise of Dr. Eduardo César Galobardes and Dr. Joan Sorribes Gomis.

Bellaterra, June 13, 2017

Author

Claudio Daniel Márquez Pérez

Thesis Advisors

Dr. Eduardo César Galobardes

Dr. Joan Sorribes Gomis

Acknowledgements

In many more ways this Ph.D. thesis reflects the support and caring of the countless people who influenced my life and this work.

I would like to express sincere thanks to my advisors Dr. Eduardo César Galobardes and Dr. Joan Sorribes Gomis for their support during the undertaking of this research. The completion of this dissertation would not have been possible without their encouragement, advises, patience, healthy discussions and committed guidance. Furthermore, I would like to extend my gratitude to Prof. Dr. Jochen Prehn of Department of Physiology and Medical Physics for his hospitality, and participation along my research stay at Royal College of Surgeons in Ireland (RCSI).

I would like to thank all the people and my fellows at the Computer Applications in Science & Engineering Department (CASE) at the Barcelona Supercomputing Center-Centro Nacional de Supercomputación (BSC-CNS) for their encouragement and support, specially to Dr. Mauricio Hanzich, Dr. Josep de la Puente, Dr. Arnau Folch and Dr. José María Cela.

I also want to thank all the members and staff at the Computer Architectures and Operating Systems Department (CAOS) at the Universitat Autònoma de Barcelona (UAB), especially to Anna Sikora, Tomàs Margalef, Anna Cortés, Remo Suppi, Porfidio Hernández, Dolores Rexachs, Emilio Luque, César Allande, Claudia Rosas, Hugo Meyer, Arindam Choudhury, Javier Panadero, Albert Gutierrez, Roberto Solar, Alejandro Chacón, Javier Navarro and Daniel Ruiz.

I want to thank my parents and brothers for their love and support. My parents, *Juan Carlos Márquez Mondaca* and *Catalina Isabel Pérez Olivares*, raised me to believe that I could achieve anything that I set my mind to and without their continuous support and encouragement I never would have been able to achieve my goals.

A special thank you to my fiancée, *Laia Parra Reguill.* Words cannot describe how blessed I am to have her in my life. She has selflessly given more to me than I ever could have asked for. I love you, and look forward to our lifelong journey.

Finally and foremost, thanks goes to Jesus Christ, my God, for the many blessings undeservedly bestowed upon me.

"The Lord is my strength and my shield; my heart trusted in Him, and I was helped; therefore my heart rejoices, and with my song I will thank Him."

Psalm 28:7

"For I know the plans that I have for you, says the Lord, plans for peace and not for evil, to give you a future and a hope."

Jeremiah 29:11

Abstract

With the emergence of agent-based modelling and simulation (ABMS) platforms intended for High Performance Computing (HPC) environments, nowadays, real systems can be more accurately modelled, analysed and simulated through including larger number of more complex agents. This leads to very complex models, resulting in a high computational burden with very high computational requirements. In this sense, simulating a complex agent-based (AB) system for realistic cases is only feasible in a reasonable time if the simulation is executed in parallel on a HPC environment. Due to its scalability and simplicity, single program multiple data (SPMD) is the dominant application structure and consists of executing the same program in all processing elements (PEs), but on a different subset of the domain. However, in complex and large AB simulations, improper data partition policies and dynamic characteristics related to the creation and elimination of agents introduce uneven computing requirements and communication overhead that delays the simulation and may propagate across all PEs. At this point, an efficient dynamic solution to readjust the workload is incredibly beneficial.

This thesis proposes a methodology that enables dynamic performance enhancements for SPMD ABMS applications and spatially-explicit AB models. The methodology introduces a tuning strategy to reduce workload imbalance problems as the simulation proceeds. This solution dynamically minimises the gaps of the computing and communication workloads between PEs. As a result, the HPC ABMS platforms will be able to process a large number of agents with complex rules as fast and efficiently as possible. The strategy adjusts the global simulation workload migrating groups of agents among PEs according to their computational workload and their interconnectivity modelling the system as a hypergraph. A hypergraph is a graph generalisation that, in this case, allows more accurately modelling groups of agents' interactions. This hypergraph is lastly partitioned using a parallel partitioning algorithm to decide a proper workload distribution.

We have evaluated our strategy using a real HPC ABMS platform, so-called Flame, simulating three real AB models Susceptible-Infected-Remove (SIR), Colorectal Tumour Growth (CTG) and Keratinocyte Colony Formation (KCF). Evaluating different aspects of our methodology, as well as an integral whole, we have obtained significant performance gains and hence an important reduction of the total execution times.

Keywords: Load Balancing, Dynamic Performance Analysis and Tuning, Agent-based Simulation, Graph & Hypergraph Partitioning.

Resum

Amb l'aparició de plataformes ABMS (Agent-Based Modeling and Simulation) per entorns HPC (High Performance Computing) els sistemes reals poden ser modelats, analitzats i simulats de formes precises incloent una gran quantitat d'agents fins i tot més complexos. Aquests models complexos generen una alta càrrega computacional i alts requisits de computació. En aquest sentit, si es volen simular aquests casos en un temps raonable, només serà possible executant-ho en paral·lel i en entorns HPC. Degut a l'escalabilitat i simplicitat, l'estructura d'aplicació SPMD (Single Program Multiple Data) és la més utilitzada, la qual consisteix en executar el mateix programa en diferents PEs (processing elements), però sobre un diferent subconjunt del domini. No obstant això, en simulacions de grans dimensions i complexes apareixen requeriments de còmput irregulars i sobrecàrrega de comunicació, degut a polítiques inapropiades de particionament de dades i característiques dinàmiques de creació i eliminació d'agents, la qual cosa es propaga per tots els PEs i endarrereix tota la simulació. Per la qual cosa, és altament beneficiós tenir una solució dinàmica i eficient per reajustar la càrrega de treball.

Aquesta tesi proposa una metodologia que permet millores dinàmiques en el rendiment de les aplicacions ABMS SPMD per models basats en agents espacialment explícits. La metodologia presenta una estratègia de sintonització per reduir problemes de desbalanç de càrrega durant l'execució de l'aplicació. Aquesta solució minimitza dinàmicament les diferències de càrrega de còmput i comunicació entre PEs. L'estratègia ajusta la càrrega de treball global de la simulació, migrant grups d'agents entre PEs segons les seves càrregues computacionals i la seva interconectivitat, modelant el sistema com un hipergraf. Un hipergraf es una generalització de grafs que, en aquest cas, permet modelar interaccions de grups d'agents de forma més precisa. Aquest hipergraf es particiona utilitzant un algoritme paral·lel de particionament per a decidir una nova distribució de la càrrega de treball més apropiada.

Hem avaluat aquesta estratègia utilitzant una plataforma ABMS HPC real, anomenada Flame, i simulant tres models reals SIR (Susceptible-Infected-Remove), CTG (Colorectal Tumour Growth) and KCF (Keratinocyte Colony Formation). Avaluant diferents aspectes de la nostra metodologia, així com en el seu conjunt, hem obtingut importants guanys de rendiment i una significativa reducció del temps total d'execució.

Paraules clau: Balanç de càrrega, Anàlisi i sintonització de rendiment dinàmic, Simulació basada en agents, particionament de grafs i hipergrafs.

Resumen

Con la aparición de plataformas ABMS (Agent-Based Modelling and Simulation) para entornos HPC (High Performance Computing), los sistemas reales pueden ser modelados, analizados y simulados de forma más precisa incluyendo una gran cantidad de agentes aún más complejos. Estos modelos complejos generan una alta carga computacional y altos requerimientos de cómputo. En este sentido, si se desea simular estos casos en un tiempo razonable, es sólo posible ejecutando en paralelo y en entornos HPC. Debido a la escalabilidad y simplicidad, la estructura de aplicación SPMD (Single Program Multiple Data) es la más usada, la cual consiste en ejecutar el mismo programa en diferentes PEs (processing elements), pero sobre un distinto subconjunto del dominio. Sin embargo, en simulaciones grandes y complejas, aparecen requerimientos de cómputo irregulares y sobrecarga de comunicación, debido a políticas inapropiadas de particionamiento de datos y características dinámicas de creación y eliminación de agentes, lo cual se propaga por todos los PEs y retrasa toda la simulación. Por lo cual, es altamente beneficioso tener una solución dinámica y eficiente para reajustar la carga de trabajo.

Esta tesis propone una metodología que permite mejoras dinámicas en el rendimiento de aplicaciones ABMS SPMD para modelos basados en agentes espacialmente explícitos. La metodología presenta una estrategia de sintonización para reducir problemas de desbalance de carga durante la ejecución de la aplicación. Esta solución minimiza dinámicamente las diferencias de carga de cómputo y comunicación entre PEs. La estrategia ajusta la carga de trabajo global de la simulación, migrando grupos de agentes entre PEs según sus cargas computacionales y su interconectividad, modelando el sistema como un hipergrafo. Un hipergrafo es una generalización de grafo qué, en este caso, permite modelar interacciones de grupos de agentes de forma más precisa. Este hipergrafo se particiona utilizando un algoritmo paralelo de particionamiento para decidir una nueva distribución de la carga de trabajo más apropiada.

Hemos evaluado esta estrategia usando una plataforma ABMS HPC real, llamada Flame, y simulando tres modelos reales SIR (Susceptible-Infected-Remove), CTG (Colorectal Tumour Growth) and KCF (Keratinocyte Colony Formation). Evaluando distintos aspectos de nuestra metodología, así como en su conjunto, hemos obtenido importantes ganancias de rendimiento y una significativa reducción del tiempo total de ejecución.

Palabras clave: Balance de carga, Análisis y sintonización de rendimiento dinámico, Simulación basada en agentes, Particionamiento de grafos e hipergrafos.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Nowadays, there are a large amount of scientific and engineering problems that can be studied and solved thanks to the computational resources existing today. Also, the reduction of the cost of computational resources leads to accessibility not previously had thereof. For instance, solutions for problems such as: weather forecasting, fluid dynamics, materials and structures simulation, and human genome analysis, can be achieved today.

The computational systems capable to support the areas mentioned above are known as High Performance Computing (HPC). The HPC environments solve computing problems when these become more complex and the amount of required computing power increases. Such problems demand extensive computation, memory, disk accesses and communications. In such circumstances, decomposing the problem (data and/or code) into a parallel program may be the only way to achieve a solution in a reasonable time [24].

In many cases, HPC applications dynamically change their behaviour during the execution. Here, the initial decomposition may not offer an efficient solution and the computing/communication workloads could have to be dynamically rearranged at runtime in order to avoid an excessive execution time. Within these kind of HPC applications, there is a particular case called Agent-Based Modelling and Simulation (ABMS) which frequently shows dynamic workload variability that negatively affects the performance of the application.

Agent-based (AB) models, also known as *Individual-based*, allow to analyse the emergent properties of a system from the interaction amongst autonomous entities called *agents* (or *individuals*)[90, 104]. Agent interactions and behaviour are defined according to their procedural rules, characteristic parameters, the whole population characteristics, dynamic characteristics of creation and elimination of agents and developments in the simulated environment. Therefore, ABMS computing/communication workload can vary dramatically over simulation time and space. At this point, an efficient dynamic solution to readjust the workload would be very helpful.

Since there are different ways to model and implement AB systems, such as agents modelled by network links [100, 50], in this thesis we consider spatially-explicit models, which means that agents are associated with a spatial location in geometric space. Spatially-explicit models are also known as agents modelled in ”knowledge space” [51] and some models describe interaction between members of adjacent social groups (with neighbouring communications) [64]. Moreover, some spatially-explicit models also can show motion patterns, i.e., agents can change their relative position in geometrical space [103, 104].

In this chapter we present a brief overview of ABMS and HPC environments. In particular, our thesis is focused on the SPMD application paradigm which is briefly explained in Section 1.2.4 and with more detail in the next chapter. We conclude with an overview of the related studies (1.4) and the presentation of the thesis proposal (1.8).

1.1 Agent-based Modelling and Simulations

Agent-based modelling and simulations (ABMS), in some particular cases named individual-based modelling (IbM), is a modelling paradigm where an emergent system behaviour is decomposed in a set of individual entities behaviour [69].

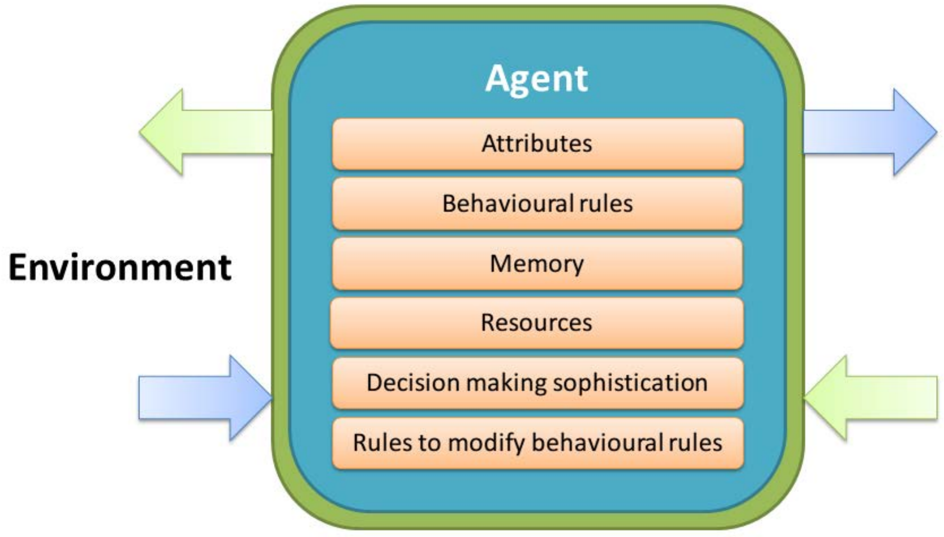
Agent
Attributes
Behavioural rules
Memory
Resources
Decision making sophistication
Rules to modify behavioural rules
Environment

Figure 1.1: Agent definition according to [69].

In this modelling, the entities are known as agents (individuals for IbM) which, for practical purposes, have properties and attributes such as:

self-directed: agents function independently of other agents and the environment (autonomous).

self-contained: agents are identifiable individuals with a set of characteristics or attributes, behaviour, and decision-making capability (modular).

social: agents have protocols to describe how they interact with other agents (interactive).

In general, the global view of the system can be achieved by replicating complex social agents' interactions, collaboration and group behaviours. ABMS is an alternative to other modelling techniques, such as equation-based modelling (EBM). The EBM approach identifies system variables and evaluates or integrates sets of equations relating these variables. One of the main examples of the use of EBM is the prey-predator model developed independently by mathematicians Alfred J.Lotka [67] and Vito Volterra [111].

Both, ABMS and EBM, approaches simulate the system by constructing a model and executing it on a computer. On one side, the ABMS execution consists of simulating the encapsulated behaviour of the agents and, on the other side, the EBM execution consists of evaluating a set of equations [83]. However, using the EBM approach for modelling the behaviour of complex social interactions, collaboration and group behaviours may be increasingly hard and might be too complex to adequately model.

On the contrary, ABMS offers better environment representation by reason of ABMS lets including a multitude of independent variables and interactions (each entity possesses its own variables and actions). Moreover, agents might include stochastic variables to implement systems with stochastic dynamics (e.g. random individual decision rules and non-trivial interactions among agents). This is useful if modelling highly non-linear systems, such as economic systems, where a good representation might be compounded by, for instance, non-linearities and randomness in individual behaviours and interaction networks, and feedbacks between the micro and macro levels. In the same way, in socio-economic systems that are inherently non-stationary, agents might introduce persistent novelty (e.g., new patterns of behaviour). In such cases, the complicated stochastic governing operations can hardly be analysed analytically and hence agent-based computer simulation is best suited for these needs.

1.1.1 Large-scale ABMS

Nowadays, thanks to the computing resources available, many complex agent-based (AB) models can be simulated in a reasonable time. Nevertheless, the computational size of the models and their resource requirements to satisfy these new demands have grown. This fact also affects the model designs, now the modellers and developers can include a large number of agents and complex interaction rules, in such cases the HPC infrastructures help to satisfy the computational requirements.

Although the AB modelling for an HPC environment may not change whatsoever respect of a sequential version, the ABMS programming does so. For an HPC ABMS application, programmers have to deal with many more problems that for a sequential ABMS, issues such as considering the architecture, operating system, or choosing the proper programming language and libraries to allow suitable problem decomposition and communication strategies. Along with these, the load balancing strategy to reallocate computing/communication workload as the simulation proceeds, to make better use of the available resources, is a key problem commonly present in most HPC application developments.

This work is focused in developing a solution to deal with workload imbalances occurring during an HPC AB simulation (Section 1.3 introduces the load balancing problem). Within this context, we studied Single Program Multiple Data (SPMD) ABMS platforms [55] using Message Passing Interface (MPI) [53] for communication (SPMD and MPI are explained further). For this reason, we briefly describe HPC infrastructures and performance issues.

1.2 High Performance Computing

The objective of parallel computing is to improve the productivity (amount of running applications per time unit) and/or the efficiency (diminish the execution time of the application). For parallelising an application, it is necessary to exploit its concurrency in order to distribute the workload among the processing elements (PEs) and thus reduce the execution time of the application. That means that parallel computing allows dividing the applications in smaller tasks to execute them simultaneously. Developing this type of applications is not simple, but the performance benefits make them essential for certain problems.

In the last decade, parallel computing has reached new areas such as: Financial, Aerospace, Energy, Telecommunications, Information Services, Defence, Geophysics, Research, etc. Developments in hardware and software for these new areas have created new challenges to be addressed, and also allowed to solve previously intractable problems.

While high-performance hardware has exhibited an immense scalability since its inception, following Moore's law or even surpassing it for a long time as a result of the outbreak of systems based on cluster, the software has been hampered for several reasons, such as the complexity of hardware systems and the complexity of its development, implementation and maintenance. To make all possible, scientists are exploring possible improvements to be made in parallel and/or distributed systems.

Unfortunately, sustain good performance indices is not simple because there are several issues that difficult reaching the optimal theoretical performance values. The reasons for such difficulties, both for development and tuning, are rooted in complex interactions between varied HPC environment components which connect the applications with the physical system, such as: system software, libraries, operating system, programming interfaces and implemented algorithms.

If a programmer wants to build a "performance model/load balancing solution", for SPMD ABMS applications, he/she could need to understand all these interactions in order to improve the performance and detect inefficiencies. This is a costly task, requiring deep study, analyse and evaluation of the application in order to establish the factors or metrics associated to such inefficiencies.

Today, we can find a lot of studies of factors that influence performance [57]. As shown in [24], we can maximise the benefits focusing on the factors associated with a particular application paradigm. Therefore, in this work we focus on recognising that factors that determine inefficiencies at application level and at the level of programming paradigms, without neglecting the different alternatives of construction of parallel systems explained below.

1.2.1 Architectural Models

There are many ways to build parallel systems, due to the variety of its characteristics they can be classified into different taxonomies. Depending on the sets of instructions and data, where a sequential architecture would be called SISD (Single Instruction Single Data), we can classify the different parallel architectures in different groups (this classification is often referred as Flynn's taxonomy) [98, 5, 74, 106, 109]: SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data), MIMD (Multiple Instruction Multiple Data) and SPMD (Single Program Multiple Data) which is a variation of MIMD.

SIMD : In a SIMD architecture homogeneous processes (with the same code) synchronously execute the same instruction on their data, or the same operation is applied over fixed/variable size vectors.

MISD : The same dataset is treated differently by the processes. It is useful in cases where many different operations must be performed on the same set of data.

MIMD : In this approach, data and task are distributed among different processors. Several execution flows (possibly different) are applied to different data sets.

SPMD : In data parallelism [55], the same code is used on different portions of data. There are multiple instances of the same task, each running the code independently as discussed in detail in Chapter 2. Due to its scalability and simplicity, SPMD is the dominant programming model for large-scale distributed-memory machines [58] and, furthermore, the most common application structure for HPC ABMS applications.

This architecture models classification is usually expanded to include various categories of computers that are not completely conform to each of these models. An extended classification Flynn is presented in [40].

Another classification of architectural models is based on control mechanisms and organisation of the memory space addresses [65, 47, 35]. Using this approach, it is possible to make a classification according to the organisation of memory (which can be physical and logical). For instance, having a physical organisation of distributed memory, but logically shared.

In the case of multiprocessors, it is possible to make a subdivision between loosely or strongly coupled systems. In a strongly coupled system, the system offers the same access time to memory for each processor. This system can be implemented through a single large memory module, or a set of memory modules so that they can be accessed in parallel by different processors (UMA). In a loosely coupled system, the memory system is distributed among the processors, providing each of its local memory. Each processor can access its local memory and to the memory of other processors (NUMA).

The increasing complexity of the applications of scientific computing and the increasing volume of information have led to the need of building systems with multiple units which include these architectural models. These systems are known as computing cluster.

1.2.2 Computing Cluster

SPMD applications mostly operate in clusters, leveraging the scalability and computing resources provided by the clusters. A computing cluster is a parallel or distributed processing system, comprised by a number of computing units (nodes) that run a number of applications simultaneously. It is a set of integrated computing resources, so-called processing elements (PEs), to run applications in less time [21].

Due to the increasing utilisation of communication networks in information systems, it has triggered a wider market and a widespread commercialisation. Consequently, the hardware used in these systems has reduced its cost so a parallel machine can be deployed at a reasonable cost. In fact, much of today's high performance systems [1] are cluster-type systems, with dedicated interconnection networks.

A collection of computers connected by a communications technology, such as (Gigabit) Ethernet, Fibber Channel, ATM, etc., are defined as cluster [5, 39]. A cluster is controlled by an administrative entity that has complete control over each end-system.

If we would classify it as an "Architectural Model", computing clusters are MIMD systems with distributed memory, although interconnection can be non-dedicated, and use simple interconnection mechanisms like a local network. Basically, a Distributed Memory-MIMD (DM-MIMD) model where communications between processors are usually slower than multiprocessors.

In a cluster, we can take advantage of idle-times of some computing units to execute other parallel applications, without any detriment to users. This also allows us to differentiate between dedicated clusters (or not), depending on the existence of users (and thus its applications) along with the implementation of parallel applications. Likewise, we can also distinguish between dedication of the cluster to run one or more simultaneous parallel applications.

An important advantage found in clusters is that they are quite scalable in terms of hardware. We could say that it is easy to build systems with hundreds or thousands of machines, commonly where network technologies are those that limit the scalability.

For allowing network scalability, the hardware must necessarily be designed for parallel processing, i.e., minimising latencies and maximising bandwidth. We often find that the base software mounted on a cluster is usually designed for specific machines, such as the operating system, which usually does not offer possibilities to have extensive control of the resources. In these cases, it is necessary to incorporate a number of layers of middleware services on the operating system, to make the cluster systems efficient. In larger scale systems, it is necessary to include a number of additional control mechanisms and management for scheduling [14] and monitoring systems, as is the case in the grid [48] and cloud [22] systems. Overall, this significantly increases the system complexity, and has a significant impact in the performance of the computing cluster.

There is currently a wide variety of hardware configurations in clusters because the construction of these has no major design constraints. Therefore, also the way to program a cluster may vary depending on which resources are going to be used or interconnected. The solution provided in this thesis is designed, evaluated and intended for improving HPC ABMS applications executed in these environments.

1.2.3 Programming Models

Parallel applications consist of a set of tasks that can communicate and cooperate to solve a problem. Because of the high dependency on the architectures of the machines that are used, and the programming paradigms used in its implementation, there is not a well-established methodology for creating parallel applications. In [47], the creation of an application is defined by four stages: Partition, Communication, Agglomeration, and Mapping (Foster's methodology).

In the Partition stage, the computing and data operations are decomposed into small tasks. During the Communication stage, the data structures, protocols, and algorithms to coordinate the implementation of the tasks are defined. Agglomeration specifies, if necessary, which tasks are combined into larger tasks with the purpose of reducing communication costs and increase the performance. In the Mapping, each task is assigned to a processor to attempt maximising processor utilisation and minimising communication costs.

These stages, can be subdivided into two groups, the first two focus on concurrency and scalability, and aim to develop algorithms that prioritise these characteristics. In the last two stages, the focus is shifted to the locality and the performance gain.

As a result of the above tasks, we must decide the appropriate programming paradigm (see Section 1.2.4), which is a class of algorithms that have the same control structure [54, 75], and can be implemented using a generic model of parallel programming.

Finally, the resulting performance is obtained combining the paradigm with the programming model over an architecture, that is the configuration of the parallel hardware, interconnection network and the available software of the system. Below, we present a classification of parallel programming models [65, 47].

1.2.3.1 Shared Memory

This model consists of a collection of processes accessing to a set of local variables and shared variables [8, 25]. Each process accesses to the shared data by reading/writing asynchronously. This model requires mechanisms to solve problems of mutual exclusion that may arise (mechanisms such as semaphores or locks) mainly as a result of concurrent access to the shared data.

In this model, tasks are assigned to threads to be asynchronously executed. These threads have access to shared memory space with the control mechanisms mentioned above. One of the most used program implementations of this model is OpenMP (Open Specifications for Multi Processing) [115], commonly used in SMP systems [56] or SM-MIMD (Shared Memory-MIMD). In OpenMP a model of parallel execution called fork-join is used, where basically the main thread begins as a single process, then at some time it performs a fork operation to create a parallel region of a set of threads. Finally, a synchronisation operation called join ends the parallel region and resumes the main thread into a sequential execution, until the next parallel region.

1.2.3.2 Message Passing

In the message passing model, programs are organised as a set of tasks with private local variables, which also have the ability to send and receive data between tasks through message exchanges. In message passing applications, processes have a distributed address space unlike shared memory applications. This is the most widely used model in HPC [65, 113].

Message passing allows greater flexibility for parallel algorithms, providing unusual control because the programmer decides how and when communications are made, that is to say, the programmer is who directly controls the flow of operations and data. It also allows the programmer to have an explicit control of the data locality, and consequently, precautions have to be taken to maintain good performance through the management of the memory hierarchy.

The programmer has a considerable responsibility in this model, the implementation should be done taking into account various aspects, such as distribution of data, communications between tasks, synchronization points and I/O operations, if any. The programmer goal is to avoid data dependencies, deadlocks and inefficient communications as well as implementing fault tolerance mechanisms [13, 2].

The most widely mean used to implement this programming model is via an API library that implements common primitives in message passing environments. Today there are several approaches [74] of libraries that support the message passing model, some of them are BSP [117] and MPI [53]. Additionally, MPI is a specification to implement message passing where the developers decide the implementation.

In this sense, there are few well-established ABMS platforms intended to run simulations in parallel/distributed environments and handle large number of agents and complex interaction rules, as explained in Section 1.4. In these cases, ABMS platforms usually provide a native support for MPI implementations in a manner that providing executions on several physical nodes.

1.2.4 Programming Paradigms

Parallel programming paradigms are classes of algorithms that solve different problems with the same control structures [54]. Each of these paradigms defines models of how to parallelise an application using a general description of how to distribute data and how is the interaction between the processing elements (PEs).

When designing a parallel application, it is important to analyse the existing paradigms in order to weight their advantages and disadvantages. This is usually because, depending on the problem type, it may be unnecessary to use a specific paradigm or, for a given problem, using a particular paradigm the performance decreases compared to another. There are different classifications of programming paradigms, but a common subset, presented in [21], is:

1.2.4.1 Divide and Conquer

In this paradigm, the problem is divided into a number of subproblems. There are three operations: divide, compute and join. The program structure is tree type, being the subproblems leaves that are executed by the processes or threads. Each of these subproblems are solved independently and their results are combined to produce the final result. In this paradigm, it is possible to perform a recursive decomposition until the subproblems cannot be subdivided further, then the solution of the problem is performed through partition operations (split or fork), computing, and joining (combine the solutions of subproblems into a solution of the original problem). It also works with tree algorithm features, such as the width of the tree, the degree of each node, the tree height and the recursion degree.

1.2.4.2 Pipeline

This paradigm is based on a functional decomposition. The application is divided into subproblems or processes, and each subproblem must be completed to start the next process, one after the other. In a pipeline the processes are called steps and each of these steps resolves a particular task. It is also important to highlight that this paradigm achieves a concurrent execution if all the steps of the pipeline are full or constantly streaming data. This is not easy to achieve, because a step output is the input of the next step. One of the most significant parameter is the number of steps because a single overloaded step usually delays the entire execution and, hence, decreases the application performance. In such situations, it necessary to apply policies for reorganising steps in order to relieve the overloaded steps and balance the workload along the pipeline.

1.2.4.3 Master-Worker

The master-worker paradigm is also known as task-farming or master-slave. This paradigm consists of two types of entities: one master and several workers. The master is responsible for decomposing the problem into smaller jobs (or responsible to divide the input data into subsets) and distribute the application tasks to different workers. Afterwards, the collection of partial results is performed in order to process the final result. The role of the workers is only receiving the information, processing it and sending the results to the master. Due its simplicity, master-worked paradigm is widespread and heavily used in traditional applications. Master-worked usually provides easy management of the workload and efficiency because the master controls the global workload distribution and, in this way, manage the workload balance among the workers. However, the master-worker paradigm suffers from scalability problems due to the master could become a bottleneck if there are too many worker outcomes and queries to process.

1.2.4.4 SPMD

In the Single Program Multiple Data (SPMD) paradigm, for a certain number of processing elements (PEs) (processes) the same code is executed but on a different subset of the input data. That is, application data are distributed between the available nodes (or machines) of the system.

The SPMD paradigm describes an iterative behaviour in which a synchronisation phase is carried out before the end of each iteration (computing phase). At this phase, the data exchange is preformed between the neighbouring nodes which depend on the nature of the problem. Due to its scalability features, given by a decentralised control structure, SPMD is the most used paradigm for large-scale distributed-memory machines and therefore for programming large-scale ABMS platforms. In Chapter 2 a more detailed analysis of this paradigm is shown, exposing its operation and the issues that affects its performance in ABMS applications.

1.3 Data Partitioning and Load Balancing

For applications where the required workload for each data section is predictable and constant over the execution time, calculating an appropriate data partition at the beginning of the execution (static load balancing) is enough to obtain a good performance.

Otherwise, to avoid imbalances during data processing and communication phases, some mechanism for adjusting the workload during runtime have to be provided. These mechanisms must decide when and how to repartition, along with calculating the new partition and redistribute the data (migrate) for the new configuration, thus, it is easy to see that achieving better performance is a very complex task.

We can mention three important aspects that need to be considered when selecting a data partitioning strategy [77]:

Load imbalance overhead : there are idle processes waiting for processes, either neighbouring or not, to complete their computing assignments.

Communication overhead : each message requires sending and receiving among the involved processes, also requires a starting time or connecting set-up time and a proportional period of time to the message or data length. In addition, each message is subject to some latency that may increase the load imbalance.

Implementation complexity : algorithm and strategy performance may vary depending on the required amount of the programmer effort. In general, the more complex the partitioning algorithm, the better result of load balancing and communication efficiency.

There are several methods for global partitioning of data among processes. A suitable choice depends on the nature of the data and performed computing [77]. In general, the simplest strategies apply to particular types of problems, while more complex strategies can be applied to a wider range of problems.

To select the best strategy for load balancing, it is necessary to know the application in order to find which partitioning strategies are better suited, and also decide whether the complexity degree of the application and partitioning allows for a static or dynamic load balancing.

1.3.0.1 Regular Partitioning

This type of partitioning is particularly applicable to data with simple geometry such as multidimensional arrays, which can be partitioned on a regular and repetitive pattern. These data can be divided by rows, columns, or blocks, as can be seen in Figure 1.2.

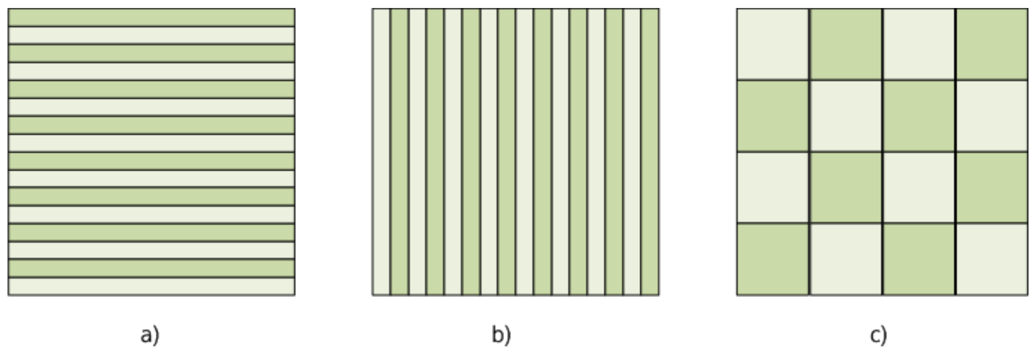

Figure 1.2: Regular partitioning examples: *a)* rows, *b)* columns and *c)* blocks.

From these three approaches, the block partitioning is often preferred when the computing phase requires data from the neighbouring processes to start the next computing phase. In Figure 1.2, for *a* and *b* partitions, each partition has one or two borders, while the partitions in c have three to four borders.

In the partitions 1.2*a* and 1.2*b*, each region has at most only two neighbours. However, each border has a large area, so this type of partition would have an aggregated communication overhead.

1.3.0.2 Irregular Partitioning

This partition type is used when doing a regular partitioning would cause a very high degree of workload imbalance. This may depend on the problem type, input data, or heterogeneous computing and communication units.

For example, in an application of air flow simulation over a airplane prototype the space can be divided as a collection of horizontal planes with different vertical distances from each other. The data and computing load of each of these resulting sections are assigned to distinct processes. Thin sections may be used in airplane sections and thicker sections where there is lower air movement (Figure 1.3 shows examples of irregular partitioning). In the resulting partitions, due to the large area of borders, this type of partitions would have a communication overhead.

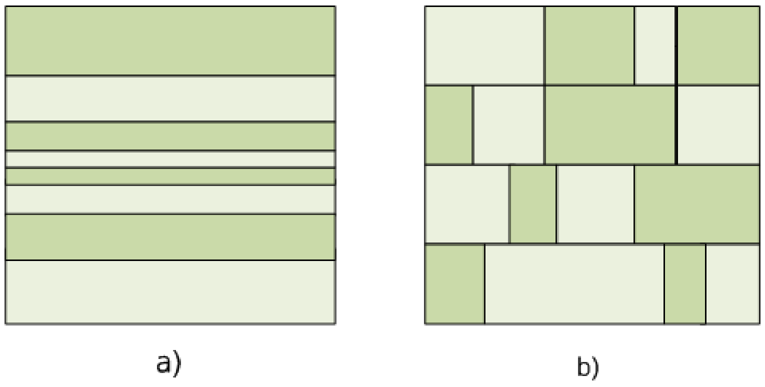

Figure 1.3: Irregular partitioning examples.

To determine the appropriate partition size, a higher level of expertise is required. First, estimating the density of the workload in each process is needed. This cannot be measured in relation to the size of the computing area, but through computational load of such area.

As mentioned above, the more complex partitioning algorithm, the better the result of load balancing and efficiency in communication. Achieving this efficiency may be limited by the complexity of communications and exchange patterns.

In addition, partitioning the data in horizontal and vertical sections generates different size areas, which could balance the computational load and minimise communication costs. In Figure 1.3*b* each section has a right and a left neighbour, and a variable number of neighbours up and down. In this case, each process must send a smaller amount of data, however, doing this is not always possible. Sometimes there are restrictions inherent to the application, as shown in the application introduced in Chapter 2.

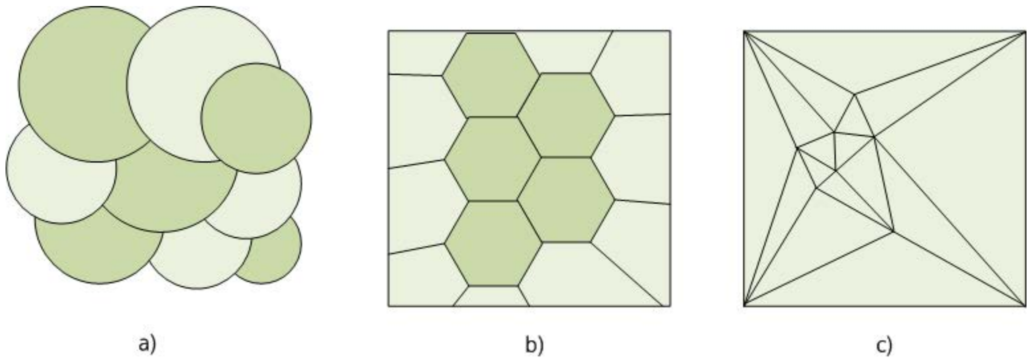

Figure 1.4: Other partitioning method examples.*a)*covering radius, *b)*Voronio diagram and *c)*Delaunay triangulation.

There are other methods of partitioning as shown in Figure 1.4. For instance, in applications based on metric spaces, the partitions can be made based on a covering radius, Voronoi diagrams, Delaunay triangulation, etc [81, 46]. In the Delaunay triangulation, each section has a maximum of 3 neighbouring sections. While in the others, the number of neighbours depends upon the number of centres, the covering radius and the scattering of elements in the space.

1.3.0.3 Random Workload

There is a group of applications in which the data location is changing during the execution, or where the location of the workload is initially unpredictable. In these cases, each element usually stores spatial information about the position of the global data using some kind of structure for this (i.e. indexing structures).

The records from a database, sorted by a particular field, are an example of this case. Here, the value of each record also stores its location with respect to the global data.

One of the main problems in such applications is the complex structures that have to be maintained to optimise the data access. Usually, in an indexing structure as more information is stored to locate data, this becomes less efficient.

In the same way, increasing the stored information the structure also grows, thus the performance of the search operations is reduced. Also, updating these kinds of structures at runtime is a big problem to deal with [44].

Certain ABMS applications fall in this classification if agents are defined with complex procedural rules and interaction patterns, variable internal workload and dynamic characteristics of creation and elimination of agents along the whole simulation. This particular behaviour provokes dynamic computing/communication workload changes that dramatically impacts the simulation performance.

1.3.0.4 Dynamic Load Balancing

One of the most critical issues when implementing an application efficiently is to choose an appropriate strategy for dynamic load balancing (DLB). A DLB is necessary, for example, when the workload cost of each computing section is unknown, the application performs dynamic creation of tasks, task migration or hardware resource variations due to external workloads [79].

As mentioned before, it is needed to decide when and how to repartition, calculate the new partition and redistribute the data for the new repartition. Additionally, it is needed to know what kind of items are going to be moved (migrated). There are several taxonomies that classify load balancing strategies, but each classification focuses on different aspects of the solution.

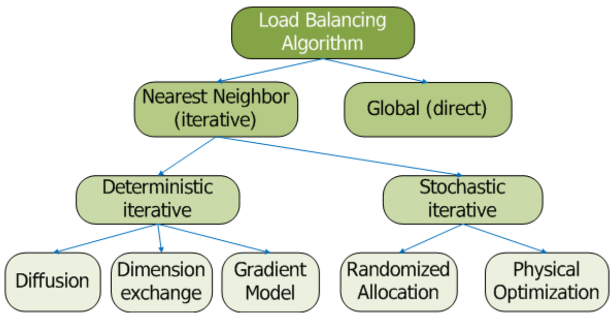

Figure 1.5: Load balancing strategies classification proposed in [26].

For instance, in [26] a classification based on consecutive local improvements in order to achieve an overall improvement (see Figure 1.5) is proposed.

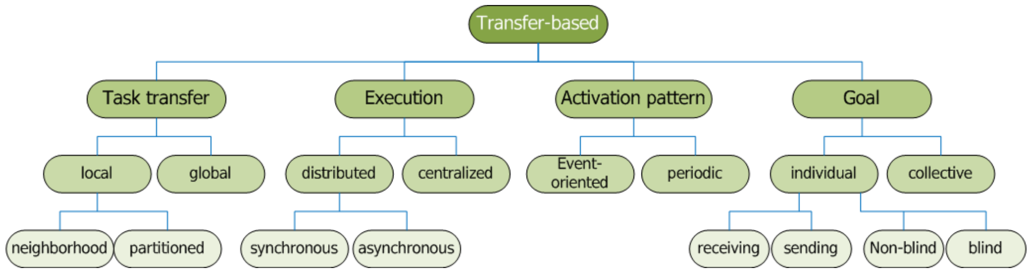

Figure 1.6: *Transfer-based* classification proposed in [86].

In [86] a set of strategies for load balancing in SPMD applications is proposed. In this classification several criteria are presented in a comprehensive manner. Figure 1.6 shows only the classification proposed in [86] for algorithms based on workload transfer when overloads or low loads are detected.

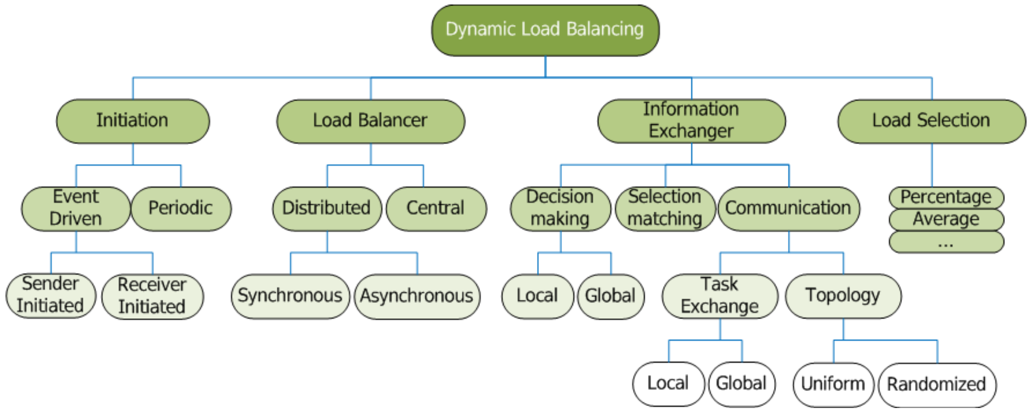

Figure 1.7: DLB taxonomy according to [79].

In the taxonomy proposed in [79], a classification based on four substrategies representing the operation of DLB is presented. In Figure 1.7 the substrategies initiation, location, exchange and selection can be seen.

It is also important to mention that the same algorithms for calculating a static partition can be used for developing a DLB strategy. Along with this, the current distribution of the workload has to be provided to calculate the new data distribution, then the data must be redistributed among different processes.

All in all, the choice of load balancing algorithm depends largely on the nature of the input data and the programming paradigm(s) involved in the application. In our case, a DLB solution is proposed which can be better categorised according to the last taxonomy (DLB taxonomy) as: *(i)* imbalance threshold event-driven initiation, *(ii)* synchronous distributed load balancer, *(iii)* local workload exchanges with randomised transference communications resulting of global decision making and *(iv)* load selection based in equally partitioning a hypergraph that represents the ABMS application workload.

1.4 Related Studies

Since this work involves a group of techniques and a wide range of research lines, we consider worth mention both the load balancing approaches in a general way, along with the intended to agent-based modelling and simulation (ABMS) and the HPC ABMS platforms in order to better understand the environment where our approach have to be deployed.

First, the load balancing strategies for HPC applications can be developed using centralised/hierarchical and decentralised approaches [86] according to the execution of the tuning decisions. The centralised approaches feature higher computational cost and scalability problems, while decentralised approaches can present problems regarding balance quality because after a neighbouring exchange the processes may possess incomplete information of the global performance. In general, many load balancing solutions have been developed such as: [119, 118, 110, 105]; but these have rarely been incorporated to a multi-purpose environment. However, we have been progressively developing a load balancing strategy for multi-purpose environments of agent-based (AB) simulations as can be seen in [70, 71, 73].

Having said that, there are some ABMS intended approaches worth to mention that face the same problem. One of these, Cosenza et al. [34], presents a distributed load balancing mechanism for ABMS based on modifying the boundaries of a global space assigned to neighbouring processors. This is a low overhead mechanism triggered in each simulation step. However, its low overhead is mainly due to the fact that it is assumed that few agents will be moved between processors. In addition, it is also assumed that all agents are of the same type, which makes it easier to decide new boundaries. The load balancing scheme we are presenting can deal successfully with different kinds of agents and it is not constrained to neighbouring processors, at the cost of a slightly higher overhead, which is compensated by the fact that the mechanism is only triggered when the imbalance threshold is exceeded.

Toh Da-Jun et al. [36] present an ABMS platform for multicellular biological systems, which incorporates a load balancing mechanism including migration of cells. However, apart for being a specialised platform, this load balancing mechanism is centralised and, in consequence, not scalable to nowadays systems.

Xu et al. [116] introduce a dynamic load balancing mechanism for an ABMS platform for traffic simulation, which presents many similarities with our proposal. They also use a graph partitioning mechanism triggered when a certain imbalance threshold is exceeded, producing the redistribution of agents among computation nodes. However, there are significant differences between both proposals. First, our approach is completely distributed, while theirs uses a master-worker approach where the master is responsible for detecting the imbalance and computing the new partitioning, and the workers are responsible for taking measurements and doing the redistribution.

Clearly, for large simulations the master may become a bottleneck. Second, our proposal optimises agent migration by packing agents (see Chapter 2), while in their system, agents are migrated one at a time. Finally, although it can be easily generalised, this proposal is specific for ABMS based traffic, while ours has been implemented in a general framework.

Also, it is worth to mention the variety of HPC ABMS platforms to be aware of their capabilities and constraints in order to understand the environment where a general purpose dynamic load balancing needs to be implemented. Having said that, several ABMS general frameworks for generating parallel simulations on

HPC environments can be found.

Repast HPC [31] was released in 2012, and written in C++ using MPI for parallel simulations. Agent types are implemented as C++ classes that are associated to contexts, which can be defined as a population of agents, and projections, which define the structure of the population contained in a context. When run in parallel, each process is responsible for executing a set of local agents. Interactions between agents assigned to different processes are managed by copying and synchronising the interacting agents in the involved processes.

D-Mason [32] is a framework written in Java, based on a master-worker paradigm. D-Mason uses idle desktop workstations subdividing the workload among these heterogeneous machines. Communication between agents is accomplished by sharing channels between workers that share information. In addition, recent improvements of D-Mason provide a load balancing schema based on executing multiple workers on the most powerful nodes.

Pandora [96] is a framework developed in C++, OpenMP and MPI. Agents are implemented as C++ or Python classes as well as the environment the agents live in (called world). Parallelisation is achieved by distributing different parts of the world among the nodes participating in the simulation. Then, each node distributes the simulation of its assigned portion among the node cores using OpenMP. The frontiers of each world partition are automatically communicated to the neighbouring nodes in each simulation step using MPI.

Finally, *FLAME* [28] allows the production of automatic parallelisable code. FLAME is written in C, it uses MPI for communication, and agents are specified using an extension of XML plus C. This is the framework used in this work, so a detailed description is given in Chapter 4.

1.5 Motivation

In the simulation research area, agent-based modelling and simulation (ABMS) is one of the most powerful simulation modelling techniques and has the capacity to provide significant benefits for studying real processes of complex systems. ABMS relies on an algorithmic description of agents that interact and simulate the expected behaviour of a system, providing a solution for defining more realistic models with complex dependencies between the entire system and the entities behaviours. Such behaviours, utilising other approaches, such differential equations, for modelling behaviours of complex social interactions, collaboration and group behaviours may be increasingly hard and might be too complex to adequately model. In these case, ABMS offers better environment representation by reason of ABMS lets including a multitude of independent variables and interactions (each entity possesses its own variables and actions).

With the emergence of ABMS platforms intended for High Performance

Computing (HPC), modellers are able to simulate models with larger number of agents and more complex rules in such a way as including more detailed models, internal processes, parameters and interactions; hence, today real systems can be more accurately modelled, analysed and simulated.

This leads to very complex agent-based (AB) models, resulting in a high computational. In this sense, simulating a complex AB system for realistic cases is only feasible in a reasonable time if the simulation is executed in parallel on a HPC environment. However, in HPC AB simulations, a weak distribution of the agents' workload may introduce uneven CPU computing and network communication overhead that delays the simulation and may propagate across all processing elements (PEs). In addition, certain AB models possess dynamic characteristics of creation and elimination of agents which dramatically introduce computing/communication workload imbalance over simulation execution. Notwithstanding, any proposal capable of reducing load imbalances and therefore, overall execution time, will have incredibly beneficial implications for running such HPC applications. At this point, an efficient dynamic solution to readjust the workload is really needed.

Considering all aspects described above, the motivation of this thesis is to propose a methodology that enables dynamic performance enhancements for ABMS applications in order to minimise the gaps of the computing and communication workloads between PEs and, as a result, enabling simulation with a large number of agents with complex rules as fast and efficiently as possible.

1.6 Objectives

The ultimate goal of this work is to design, implement and evaluate a performance improvement methodology for SPMD ABMS applications with spatially-explicit agents with neighbouring communications and variable behaviour. We address this problem by designing a methodology that dynamically monitors, measures, controls and minimises the agents computing and communication variations introduced among the PEs. Consequently, the methodology dynamically balances the workload and improves the application performance.

With this scope, we have developed the following specific objective this work as follows:

- Conduct a study on the general characteristics of HPC ABMS applications and SPMD program structure, to identify those factors that dynamically affect their performance.
- Identify the performance affectations that can be modified dynamically, taking into account results obtained in the previous point, to identify possible solutions to performance problems in SPMD ABMS applications.
- Design and implement a dynamic performance analysis, evaluation and

tuning methodology capable of improving performance of SPMD ABMS applications, in terms of execution time.

- Conduct an analysis of the problems caused by intrinsic ABMS performance affectations such as agent creation/elimination, agent variable computing/communication workload, and, in addition, the platform associated performance affectations such local/global PEs workload imbalance, monitoring, agent migration and agent message management; and discuss the effectiveness of proposed solutions to the aforementioned problems.
- Analyse the proposed methodology to avoid possible performance degradations and propose solutions (if necessary).
- Evaluate the behaviour of solutions proposed through the methodology by simulation-based experimental study; and execution of real AB models in real ABMS platform, designed and selected to confirm the proper functioning of the proposal.

1.7 Contributions

The contributions of this work are focused on achieving the ultimate goal outlined in the previous section. For this purpose, we have designed and implemented a methodology for dynamically improving performance of SPMD ABMS applications. The methodology introduces a strategy to reduce the gaps of the computing and communication workloads between PEs as the simulation proceeds. The methodology adjusts the global simulation workload migrating groups of agents among the PEs according to their computation workload and their message connectivity map modelled using a hypergraph. The hypergraph is lastly partitioned to decide a proper workload distribution. As a result, the methodology reduces the total execution time of HPC ABMS applications, and increases the efficient use of computational resources.

In the design of the methodology, we have considered:

- ABMS applications with spatially-explicit models with neighbouring communications.
- ABMS applications designed under the Single Program Multiple Data (SPMD) structure.
- Existing parallel hypergraph partitioning tools to efficiently assist tuning decisions.
- PEs allocate separate MPI processes in such a way that independents PEs operate a unique MPI process during the entire simulation.

In this way, it is possible to execute faster AB simulations with large number of agents with complex rules. Other contributions included in this work that we consider significant to mention are:

- A description of the components required to define a dynamic Load

Balancing (DLB) approach in an ABMS application.
- A monitoring and evaluation schema that measures the parallel application workload at runtime to identify performance problems in the PEs.
- An efficient agent system representation (ASR) definition through a grid-based spatial organisation that characterises the agent locations and workloads in relation to their PE.
- A weighted hypergraph based point of view of the agent workload (vertices) and interactions (hyperedges) which provides the analysis and identification of imbalances spots across the simulation space.
- An agent migration schema that minimises its runtime impact through migrating groups of agents, minimising the number of transferences and reducing the communication latency by communication/computing overlapping.
- A suitable message management definition based on the ASR definition that enables determining the internal PE messages and the messages that need to be dispatched to other PEs.
- A detailed implementation of a DLB methodology in a real HPC ABMS platform (Flame) and tested with three real AB models Susceptible-Infected-Remove (SIR), Colorectal Tumour Growth (CTG) and Keratinocyte Colony Formation (KCF).

1.8 Thesis Outline

According to the objectives and research method described above, the outline of the remaining chapters of the work is as follows.

- **Chapter 2: Load Balancing in SPMD ABMS applications**. Introduces and discusses concepts related to performance analysis and tuning of SPMD ABMS applications. In addition, more detailed description of the SPMD paradigm and ABMS application are also introduced.

- **Chapter 3: Methodology Description**. In this chapter, we present the proposed performance improvement methodology, specifically designed for SPMD ABMS applications for spatially explicit models.

- **Chapter 4: Methodology Implementation**. This chapter describes a real HPC ABMS platform as testing scenario and provides description of integrating our proposal within this real platform.

- **Chapter 6: Conclusions**. Concludes the work and presents the further work and open lines for the performance improvement methodology for HPC ABMS applications.

The bibliography completes the document of this work.

Chapter 2

Load Balancing in SPMD ABMS applications

Agent-based (AB) modelling has already been widely used for the exploration of varied research areas and in many instances modelling complex systems. However, in many cases, these simulations show severe performance problems leading to load imbalances and inefficient use of available resources. Given the nature of AB simulations, uneven CPU computing and network communication are related to the dynamic behaviour of the agents, resulting from the agents' internal state changes and external interaction rules. Moreover, performance problems can arise from the way of handling the agent states and interactions by the simulation platform in itself.

In general, agent-based modelling and simulation (ABMS) applications show significant variations in the amount of computing and communication time as the simulation proceeds. The causes of these variations are given by the large amount of interactions among agents, and the variety of behaviour rules exhibited by most of these models. Additionally, the evolution of the simulation may produce dynamic changes in the workload since global agent's behaviours may introduce dynamic changes in the agent rules. Therefore, during a parallel execution of the simulation, load imbalances are likely to appear.

Furthermore, since the large number of agents and complex interaction rules, ABMS require HPC (High Performance Computing) infrastructures to satisfy the computational requirements. In these cases, the large number of agents produces even more computing imbalances and enormous amount of messages. Consequently, these issues introduce unfit synchronisation phases which degrades the global performance of the application resulting in an increase in the completion time of the simulation steps. These issues need to be studied and mitigated as soon as possible to achieve the desired performance enhancements.

Accordingly, correcting these issues involves dynamically handling computing and communication workloads. For this reason, to manage the computational size of the models and dynamically adjust computing/communication workload, a load balancing strategy is needed. The load balancing strategy will be the responsible for making better use of the available resources through automatic workload reconfiguration mechanisms. Summarising, the load balancing strategy will keep controlled and minimised computing and communication imbalances and, as a result, enabling simulations of complex and large AB models in a reasonable time.

Within the HPC ABMS world, there are few well-established platforms intended to run simulations in parallel/distributed environments and handle large number of agents and complex interaction rules.Some of the most relevant platforms (described in [95]) are RepastHPC [31], D-Mason [32], Pandora [96] and Flame [28] which provide a native support to, among other features, collaboration between executions on several physical nodes and the distribution of agents between processing elements (PEs).

RepastHPC uses the same concepts as the core of RepastSimphony [78], that is to say, it uses the concept of projections (grid, network) but this concept is adapted to parallel environments. The agent models can be implemented using C++ or ReLogo (a derivative of the NetLogo). The communications are handled using MPI and Boost library [99].

D-Mason is the distributed version of the Mason platform designed to overcome its limitations for large number of agents and facilitate parallel simulations without rewriting Mason model codes. D-Mason uses the Java language to implement the agent model and ActiveMQ JMS as a base to implement communications.

Pandora is able to treat thousands of agents with complex actions and run simulations using geographic information system (GIS) for spatial coordinates. Pandora uses the C++ language to define and to implement the agent models and automatically generates MPI code for managing the communications.

Flame simulation is based on the definition of finite state automata with memory referred to as X-Machines [30], which are able to send and receive messages at each state. The agent models are defined in a combination of C and XMML language (extension of the XML language designed to define X-Machines) and the parallel communication is carried out through the automatic generation of C/MPI code.

According to [95], these platforms have been designed to target high performance computing systems such as clusters, whereas others are more focused on distribution on less coupled PEs such as a network of workstations. Furthermore, the most used paradigm is SPMD (distributed using autonomous processing elements) and only D-MASON uses a master-worker paradigm.

Finally, the most important ABMS platforms intended to HPC architectures are programmed using the SPMD paradigm. These ABMS platforms have to deal with imbalance problems inherent to AB simulations nature. In general, agent based applications show significant variations in the amount of computing and communication time as the simulation proceeds. Therefore, defining and finding a proper general solution that improves the SPMD ABMS applications performance is clearly necessary.

For this reason, this thesis presents a general policy that dynamically balances the computational load of the simulation considering also the amount of communication among the PEs.

This section introduces and discusses concepts related to performance analysis and tuning of SPMD ABMS applications. In addition, a detailed description of the SPMD paradigm and ABMS application intended to HPC environments are also introduced.

2.1 SPMD applications in HPC

Most parallel applications are designed under the SPMD paradigm due to its scalability features given by a decentralised control structure. In the SPMD paradigm each processing element (PE) executes basically the same code, but each PE computes a different section of the application input data [101]. Therefore, a distribution of the input data is performed among the available PEs.

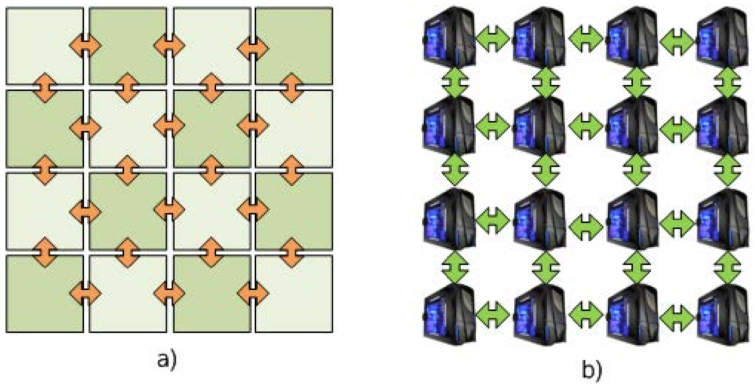

Figure 2.1: Data distribution and communication between neighbouring PEs in SPMD.

This structure generally allows the data to be distributed even among the PEs (see figure 2.1*a*)), where each PE is responsible for a defined portion of all input data. The communication usually occurs between neighbouring PEs as shown in Figure 2.1*b*. The amount of data to be communicated should be proportional to the size of the boundaries/data-dependencies of the computed section, while the computing workload should be proportional to the volume of the problem.

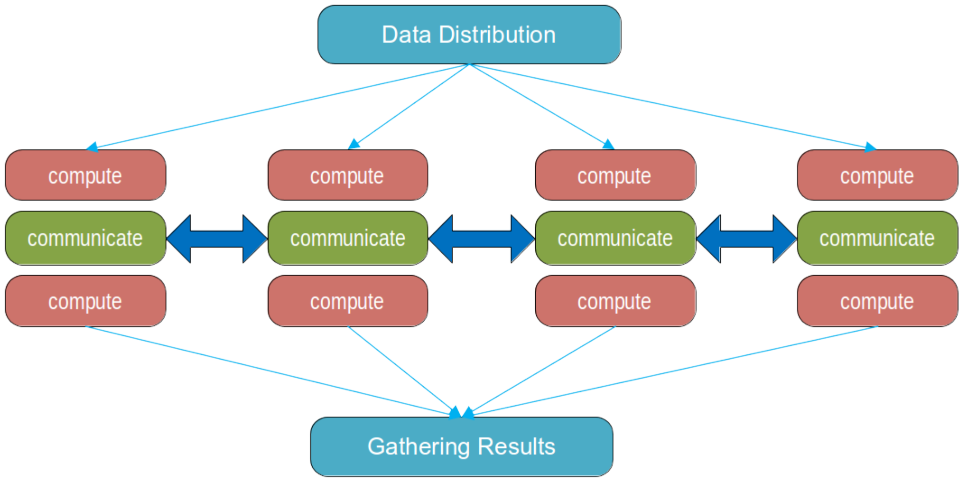

Figure 2.2: Schematic representation of a SPMD application.

Commonly, SPMD applications have an initialisation phase where the input data is read from disk, and an initial distribution of data to each PE or a self-generating of the input data is performed. Figure 2.2 shows a schematic representation of this paradigm. Also, depending on the problem, the parallel application may periodically perform global synchronisations among all PEs for gathering results or coordinate computing.

Therefore, SPMD applications have phases of computing and information exchange (communication). These phases are repeated until the end of execution, being separated by synchronisation logic points which are commonly called iterations. Generally, the synchronisation phases only apply for data dependencies and step explicit messages to satisfy these dependencies.

Figure 2.3 shows two iterations, each consisting of a computing and exchange phase. With regard to the communications, the degree of data dependencies among the PEs determines the complexity of communications during the execution.

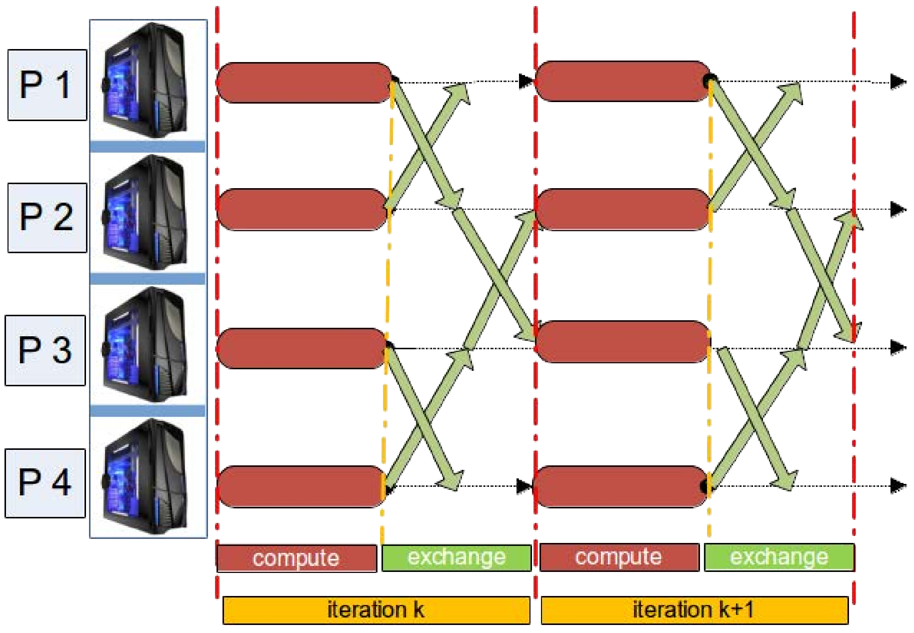

Figure 2.3: Computing and exchange phases within an iteration.

Usually, when hundreds or thousands of PEs are used, a special emphasis on programming is required, since the application have to be well-structured to avoid load imbalance and deadlock problems [16]. In Addition, if the workload in each phase is very different among the PEs, the cost of each phase is increased, since this paradigm is very sensitive to any PE workload imbalance. Usually, a single imbalanced PE is enough to cause a general level blockade that delays the resolution of the problem (this will depend on the application, it could be global or local), hence none of the PEs may move beyond the next global synchronisation point.

In general, SPMD applications are typically very efficient when the environment is homogeneous and the data are well-distributed over the PEs. Consequently, the data distribution may significantly impact the application performance, especially if the application has irregular characteristics, wherein the amount of workload changes during execution of the application [104, 97, 12] as on most ABMS applications.

Therefore, when the computing workload or capacity of the different PEs is heterogeneous, SPMD applications require the support of load balancing strategies to be able to adapt the workload distribution during the application execution. In the same way, SPMD ABMS applications require an adequate data partitioning and load balancing policies to keep homogeneous workload as the simulation occurs.

According to the literature, most HPC ABMS platforms are designed under the SPMD paradigm and do not include load balancing mechanisms. On the contrary, most of load balancing studies take place usually in non-SPMD platforms (commonly master-worker platforms), or usually the strategies are bound up to the nature of the agent model. With this in mind, we focus on ABMS platforms that run on parallel distributed environments under the SPMD paradigm (SPMD ABMS).

2.2 Load Balancing in SPMD ABMS

Following up on this idea, the performance problems of ABMS applications have their roots in the emergent behaviour of the agents that are dynamically changing during the simulation. In accordance with this emergent behaviour, the agents generate heterogeneous computing workload and large amount of varying communications. On the one hand, the computing workload varies in accordance with the agents' dynamic internal state and the changes in the amount of agents. While the communication workload, on the other hand, raises when the number of agents dynamically increases and when the agents communicate to agents belonging to other processing elements (PEs); thus increasing the number of PEs, this external communication will also likely increase.

Hence, the dynamic agents' organisation plays a critical role in the resulting performance of the application because it impacts directly over the communication dependencies. Since the changes in the ABMS performance are dynamically generated by the development of the model execution, the load balancing solution has to be also addressed dynamically. Taking into account all this, instinctively when an ABMS application requires computing and communication workload reconfiguration, the solution should be likely addressed into agent migrations constrained by reducing the external-PE agents' dependencies.

This also means that, regardless the load balancing approach, the agents computing and communication variations introduced among the PEs need to be monitored, measured, controlled and minimised. Considering all these issues the approach should manage the PEs with excess of computing and communications caused by excess of agents and messages among the PEs.

Having said all that, we conclude that a load balancing mechanism may require to include a proper method to know the agent locations with respect to the PEs, identifying the agents' communication dependencies and migrating agents between PEs. Finally, we need a set of rules to make these elements work together with the purpose of performing global workload reconfigurations while the communication is minimised.

2.3 Load Balancing Components

The agents are constantly moving and changing along with the simulation progress. In addition, during the system progress, agents might be able to disappear or reproduce or simply perform more computing/communication workload. These behaviours contribute to generate unexpected workload imbalances. Furthermore, after executing an agent migration operation the overall status of the simulation changes as a result of the agent relocations across the processing elements (PEs).

Therefore, identifying the agents involved in these dynamic workload behaviours would help to make better and more accurate reconfiguration decisions. For this reason, we consider that achieving a proper representation of the agent simulation system is a fundamental part of the entire load balancing process. Then, the reconfiguration decisions may command migration of agents based on the information retrieved from the agent system representation.

2.3.1 Agents System Representation (ASR)

The ASR is the mean by which the load balancing will understand the global and local behaviour of the agent in the simulation. The ASR has to provide information to diagnose the workload imbalance spots and help to identify the relation between such computing and communication volumes with the agents. It has also to allow retrieving agent spatial locations and their locations across the PEs.

Additionally, the information has to help to determine the agents' communication relations in order to perform proper migration decisions. In the same way, the design of the ASR has to take into account providing the required information for partitioning mechanisms.

Furthermore, building the ASR has to be feasible in runtime. This means that the construction of such ASR should not stop the application execution and neither have excessive CPU and memory requirements. Likewise, the stored agents' performance information should not be too large in such a way that it cannot be handled. All these points usually become particularly important when scalability is desired in large and complex simulations.

2.3.2 Agents Migration

By means of migrating agents features, the load balancing must be able of transferring agents from a PE to another in order to modify the workload of such PEs and, consequently, affect the global workload distribution of the simulation.

Migrating agents is not an easy task; not only it involves sending and receiving data over the network, but also stopping the simulation until all the relocated agents are totally reconnected with the platform components involved in the simulation. Basically, the agents need to be disassociated from their original PEs, and subsequently reconnected to their recipient PEs. Eventually, the new agents distribution has to be communicated to the implicated PEs.

As for the ASR, the migration operation has to keep low overhead to reduce its impact over the runtime. The performance of a migration operation is directly affected by issues such as the size and number of agents to be migrated, the number of involved PEs, and in second instance by the performance of the algorithms involved in handling the entire migration.

As efficiently managing agent migration can be complex, nowadays there are a few parallel simulation environments that include some features intended to migrate agents, but they normally are not truly focused on supporting agent migrations or some sort of workload reconfiguration strategy.

In general, an agent migration mechanism is necessary to implement policies for workload redistribution. In addition, it should be noted that a workload reconfiguration criterion must be established to decide which agents should be sent and when the migration should be performed.

2.3.3 Tuning Decisions

Nowadays, load balancing has been studied extensively in distributed systems, but the intelligence used for reconfiguring the workload and improving the performance varies according to the problem nature. This is rooted in the wide range of parallel applications and problems. Additionally, the different methods used for correcting the imbalance problems can also vary according to the trade-off between various issues such as performance goals/targets, the expected gain, the overhead for executing the load balancing, etc. For these reasons, it is very difficult to generalise the tuning decisions and solutions applied to a wide range of applications.

Apart from that, in the ABMS case, it is very important to recognise/identify/study the nature and implementation way of the AB model, because these may establish a different set of computing and communication patterns among the PEs.

Notwithstanding these difficulties, we can define some key aspects to establish appropriate rules to deal with the imbalance problem. In conjunction with defining these rules, the measure points and the information required for executing such decision will appear.

In the tuning decisions, the way of monitoring the agents may directly affect the application performance. In these sense, the way of measuring workload has to be defined including if we capture and store measures, for example, by individual agents, groups of agents, agent' types, spatial blocks, PEs, etc. Moreover, we must also define which computing/communication metrics are of interest.

Additionally, choosing a centralised or decentralised load balancing approach may severely restrict the measure points, comparison procedures and decision criteria. Along with this, it must be determined if the decisions are taken and executed by all the PEs or groups of them. Summarising, we must decide the level of global performance knowledge required, under what performance conditions the load balancing is executed and based on what criteria.

Considering such aspects, a set of reconfiguration decisions can be established to enhance the application performance. However, as it was mentioned in the previous section, it is very important to take into account the overhead introduced by such decisions in order to build an efficient load balancing strategy.

2.4 Summary

Throughout this chapter, we have shown the SPMD paradigm operation because most parallel applications are implemented with this structure due to its scalability features. In the same way, large scale ABMS platforms are designed under this paradigm (SPMD ABMS). Therefore, to implement a load balancing solution for SPMD ABMS, understanding the advantages and inefficiencies of SPMD paradigm is essential.

Additionally, to achieve good performance, it is required to consider many aspects, such as: communications, problem nature, data dependences, the internal structure of storage, programming paradigm, HPC architecture, etc. For this reason, there is not a single solution to fit all the imbalance problems. Because of this, the load balancing strategies can be developed varying the problem point of view and solution scope.

In general, we have concluded that all the tuning decisions included in a load balancing strategy for SPMD ABMS applications have to be performed dynamically because the agents behaviour variability. Likewise, we have separated the components into three categories: Agents System Representation (ASR), Agents Migration and Tuning Decisions.

The ASR provides the related agent information with their associated performance features/metrics and location across the processing elements (PEs). The Agent migration consists of everything related to transferring agents between PEs. Lastly, the Tuning Decisions comprise all the rules required for deciding a global workload reconfiguration of the application.

Chapter 3

Methodology Description

Efficient simulation of large number of agents with complex interaction rules in High Performance Computing (HPC) systems represents an important challenge. Nowadays, thanks to the computing resources available, many complex agent-based (AB) models can be simulated in a reasonable time. Nevertheless, these applications show severe performance problems mainly due to load imbalances, inefficient use of available resources, and improper data partition policies. Moreover, the variable workload, produced by the emergent behaviour of the agents, usually introduces performance inefficiencies across the parallel system at runtime.

Most performance problems in agent-based modelling and simulation (ABMS) parallel platforms are related to the intrinsic dynamic workload variability throughout the simulation execution. In these cases, these platforms require load balancing (LB) strategies to reconfigure the workload. Usually, these strategies try to detect the application imbalance, calculate a distribution of agents that would correct the problem, and perform the corresponding migrations of agents.

For calculating a balanced distribution of agents, a dynamic load balancing (DLB) strategy must be able to predict the performance of the simulation for different agent distributions. This prediction could be a very difficult task in simulations with large number of agents with complex interaction rules.

The main goal of this thesis is to propose a novel strategy that enables dynamic performance enhancements for HPC agent-based applications with agents defined with a relative spatial position in geometric space, namely spatially-explicit models [51, 103, 104]. The proposed methodology tunes dynamically the execution of agent-based applications redistributing the agents for minimising computation imbalances, but taking also into consideration the communication among agents in order to reduce the communication overhead. As a result, a HPC agent-based platform will be able to simulate a large number of agents with complex rules as fast and efficiently as possible.

The methodology introduces a strategy to reduce imbalance problems as the simulation proceeds. The methodology adjusts the global simulation workload migrating groups of agents among the processing elements (PEs) according to their computation workload and their message connectivity map modelled using a hypergraph. A hypergraph is a graph generalisation that, in this case, allows more accurately modelling agent system interactions. This hypergraph is lastly partitioned using a parallel partitioning algorithm to decide a proper workload distribution.

In this thesis, we present the methodology defined to dynamically improve the performance of ABMS applications developed with the SPMD paradigm, but we do not disregard that such methodology can be extended to others paradigms.

The rest of the chapter is organized as follows. The following section (3.1) discusses relevant issues involved in LB strategies for ABMS applications and introduces important concepts associated to our strategy. In Section 3.2, the main characteristics of the performance improvement methodology for HPC ABMS applications are described. Finally, the performance improvement methodology is summarised and discussed in Section 3.3.

Before proceeding with the methodology explanation, it is necessary to discuss the initial assumptions about developing LB strategies for HPC ABMS applications.

3.1 Discussion

Generally, to facilitate the parallel execution of ABMS applications, the input agent population is divided into smaller groups that can be processed separately. However, in many cases, these applications/platforms show severe performance problems mainly due to load imbalances, inefficient use of available resources, and improper data partition policies. At the same time, the roots of these performance problems can be grounded on the dynamic behaviour of the agents; thus, this makes even harder to effectively tackle the simulation performance problems.

In HPC agent-based applications, the load imbalances are commonly remedied by redistributing the agents to obtain a similar computational workload and reduce the communication overhead for all processing elements (PEs). In these cases, to achieve a better workload adjustment, the agents migration should keep together the agents that work and communicate more frequently for the purpose of reducing the external-PE agents' communications and dependencies. In this way, the reconfiguration will impact the computing as well as the communication workload. However, to develop an efficient strategy to reconfigure the workload according to this principle, the agents' communication relations and computing workloads need, in first place, to be traced, analysed and understood.

In order to understand these agent dependencies, the way of representing the agent interactions, called Agent System Representation (ASR) in previous chapter, plays a significant role in the LB approach. In fact, depending on the accuracy and complexity of the ASR, the tuning decisions might be more or less sophisticated and effective. Intuitively, it should be evident that having more detailed information about the agents' behaviour, platform and PEs performance, will enable the strategy to better diagnose the root causes of the imbalance problems and, therefore, to apply better tuning decisions.

However, a dynamic strategy should be able to take decisions efficiently, i.e., in a short time, and minimising the intrusion and number of required resources, building and utilising an accurate and complex ASR may significantly increase the strategy execution time and its resources requirements.

For instance, [70] proposes a schema that dynamically adjusts the computational workload and migrates agents, notwithstanding, the communication among PEs is not considered. In this case, a minimalist ASR is built considering the joint agents workload and the quantity of agents for each PE. Due to the lack of detailed information about the agents communication and imbalance spots, the dynamic load balancing (DLB) has low overhead when balances the computing workload but totally forgets the agents communication dependencies and, therefore, in many cases, it substantially increases the amount of messages among PEs.

On the other hand, building an ASR representing each particular agent and message would be extremely accurate and would allow to determine the optimal distribution of agents in terms of computational and communication workload. However, the amount of memory and time needed to build this ASR would unaffordable for any simulation involving thousands or millions of agents. In such a case, in term of resources, the CPU/memory requirements for storing this information may also become unmanageable when a large number of agents with complex interaction rules are simulated.

Additionally, in some cases, the AB simulation may perform massive creation of new agents leading to an indefinite domain expansion, while, in other cases, the elimination of agents may shrink the domain. Therefore, the ASR must be flexible, so it can be efficiently adapted to these changes in the agents' domain.

According to these ideas/notions, we introduce a dynamic load balancing methodology that tunes the global simulation workload by migrating groups of agents, simplifies the ASR construction requirements using a mixed clustering-rastering strategy, understands the simulation through a hypergraph-view that brings a more suitable agent system representation than graphs, and, finally, makes tuning decisions based on a hypergraph partitioning algorithm. In order to clarify the methodology overview, before fully describing the approach, we have considered convenient to briefly introduce certain concepts associated to our solution.

3.1.1 Clustering

Clustering is a technique for exploratory data analysis, widely used in applications ranging from statistics, computer science, biology or social sciences or psychology [112]. Informally, clustering is a division of data into groups of similar objects[10], where similarity is determined by some provided function or sort criterion [52].

Formally, given a set of n objects, the process of clustering partitions the set into unique subsets such that the objects in each subset share specific common characteristics between themselves and dissimilar to characteristics of other groups. These common characteristics can usually be specified as some mathematical relation and the objects can be viewed as points in a n-dimensional space. In terms of geometry, clustering divides the points into groups according to their spatial location [52]. Figure 3.1 shows five clusters of a set of points in a two-dimensional space (each rectangle represents a cluster). In accordance with the geometric point of view, points in the same cluster are somewhat closer in terms of their Euclidean distance to one another than to points in other clusters.

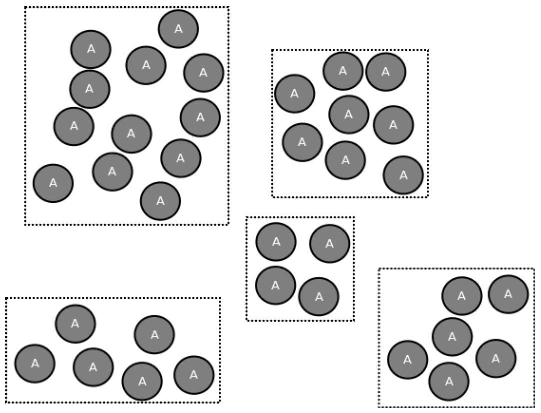

Figure 3.1: A set of points that is partitioned into five subsets [52].

Thanks to clustering, applications and algorithms can work with a small amount of data, manageable set of groups and, to a lesser degree, easier data processing [52]. Nevertheless, clustering data necessarily implies losing certain fine details, but achieves a simpler representation [10]. There are several clustering algorithms in the literature, but they can be broadly categorised into Partitioning-based, Hierarchical-based, Density-based, Grid-based and Model-based [41]. In this work, the ASR uses a Grid-based algorithm which divides the space into cubic regions (grids). In general, Grid-based clustering algorithms have fast processing time and its performance directly depends on the size of the grid, which determines the resulting number of groups from the spatial division. However, the cluster construction and utilisation would collapse the CPU/memory resources when processing a large simulation space and very small grid divisions.

In this sense, some AB models perform constantly random creation/elimination of agent resulting in dynamic expansion/shrink the simulation space (space occupied by agents). This behaviour requires techniques intended to simulate an ever expanding/shrinking space and variable amount of agents. In these cases, the ABMS platforms must be assisted by techniques for minimising the simulation domain covered by grids and allowing the dynamic spatial expansion of the simulation domain. In this manner, the simulations can avoid early workload volume limitations and lack of CPU/memory resources.

3.1.2 Domain Trimming

For the purposes of this work, we call Domain Trimming (DT) the method for determining the areas of the whole simulation domain that are occupied by agents. In this way, the ASR only includes these areas and, consequently, the performance analysis and tuning will only consider the portion of the domain occupied by agents. In addition, this method allows for the adaptation of the ASR to constantly/randomly growing and shrinking domain. This idea is taken from rasterisation techniques which, for a given geometric primitive shape, figure out which pixels the primitive covers for a perspective projection. The primitive is converted to a two-dimensional image where each point of this image contains such information as colour and depth. This technique, in part, determines which pixels/squares/zones of an integer grid are occupied by the primitive. Thus, the resulting shape determines the visible areas to be processed and the hidden areas outside to the projection [85].

On the basis of this idea, the spatial domain can be explored to determines which is the actual domain or areas covered by agents, create the ASR and divide the spatial space in accordance with these areas and, on the other, discard the empty areas and the spatial limits of the simulation. Similar approaches have been designed in 2D particle simulations [11]. Proceeding in this manner, it is possible to focus the available computational resources on processing these occupied areas and model an indefinitely large domain which randomly grows and shrinks.

3.1.3 Graphs & Hypergraphs

Graphs theory is a major and very popular branch of mathematics concerned with combinatorics [17]. It is highly utilised to model many types of relations and abstract complex systems into a simplified representation [91]. In general, the graphs theory is an important mathematical tool in a wide range of subjects in diverse fields, such as computer science, physics, biology, or social systems [114].

Formally, a graph *G=(V,E)* is a finite and non-empty set of vertices (or nodes) *V* and a set of edges (or links) *E.* If the edges are ordered pairs *(u, v)* of vertices,

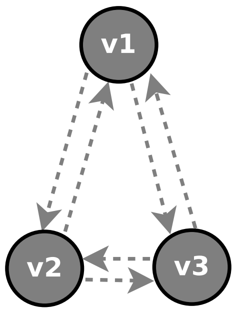

Figure 3.2: Graph example with three vertices ($v1, v2$ and $v3$).

then the graph is said to be a *directed graph*, and u is called the *tail* and v is called the head of the edge *(u, v)*. If the edges are unordered pairs (sets) of distinct vertices, also denoted by *(u, v)*, then the graph is said to be *undirected graph.* Hereunder, we introduce some terminology that is relevant to understanding this thesis.

(1) The number of vertices in a graph *G=(V,E)* is denoted by $|V|$, which also defined the *order* of the graph, that is say, $|V|$ measures the size of V by its number of vertices.
(2) If there are weights associated with the edges then the graph is a *weighted graph* [3].
(3) Two vertices u and v are *adjacent* if they are connected by an edge, in other words, *(u,v)* belongs to *E.*
(4) From this, when the graph is undirected the adjacency relation is symmetric, but when the graph is directed the relation is not necessarily symmetric [33].
(5) The *degree* of a vertex, also written as *d(v)*, is defined as the number of vertices adjacent to it.
(6) A *subgraph* of a graph G is a graph, each of whose vertices belongs to *V(G)* and each of whose edges belongs to *E(G)* [114].

A hypergraph H on a set of vertices V is a pair *(V,E)* where E is a set of non-empty subsets of V called hyperedges such that $\bigcup E = V$. This implies in particular that every vertex is included in at least one hyperedge. In doing so, a graph is a particular case of simple hypergraph where every hyperedge is of size 2 [19]. Unlike graphs, the number of vertices and hyperedges may differ, making the model suitable for complex combinatorial problems.

In a hypergraph more than two vertices may be linked, so the edges (named hyperedges) of a hypergraph are (arbitrary) subsets of the vertex set. A standard reference of this theory is due to [9]. The hypergraphs generalise standard graphs by defining edges between multiple vertices instead of only two vertices. Hence

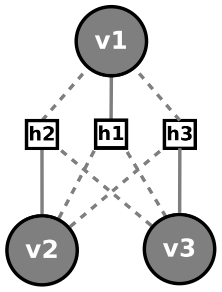

Figure 3.3: Hypergraph example with three vertices ($v1, v2$ and $v3$) and three hyperedges ($h1, h2$ and $h3$).

some properties must be a generalisation of graph properties and many of the definitions of graphs carry verbatim to hypergraphs [18].

Especially in computer science, graph theory is widely applied in research areas such as data mining, image segmentation, clustering, image capturing, networking, etc. [102]. Due to their simplicity and generality, graphs are powerful tools for modelling complex problems, even becoming one of the pillars of theoretical computer science [62]. In this context, graphs help in solving domain dependent optimisation problems modelled in terms of weighted or unweighted graphs [84].

Hypergraph has become increasingly popular over the last decade because it provides a better representation for certain problems, for instance modelling data exchange, but the algorithms for processing hypergraphs are usually slower than the corresponding ones for graphs [89].

For complex problems, graphs may not fit in the memory or cost too much to partition. In these cases, graphs algorithms could be parallelised to obtain results in a reasonable time. Several graph processing libraries has been developed with the objective of assisting applications. Some of these libraries offer parallel versions of the graph processing algorithms, such as ParMetis [60], PT-Scotch [27], and Zoltan [38]. However, even with these libraries, parallel graph algorithms (and in particular, hypergraph partitioning) are notoriously difficult to implement [62].

3.1.4 Graph/Hypergraph Partitioning Algorithms

Graph partitioning, also known as *k-way* partitioning, is one of the most relevant problems in graph theory, also a well-studied problem in combinatorial scientific computing, and extremely important in parallel computing. Since a hypergraph is a generalisation of a graph wherein edges can connect more than two vertices and are called hyperedges [80], the following concepts are only explained in term of graph but we consider they can be also extrapolated to hypergraph.

The partitioning problem can be applied to weighted or unweighted graphs, and can be stated as finding a given number of vertices subsets, which meet the following requirements: (i) the cardinality of all subsets is the same and (ii) for each vertex in a subset, the number of adjacent vertices belonging to other subsets is minimal [6]. Graph partitioning is an NP-hard (non-deterministic polynomial-time hard) problem and solutions are generally derived using approximation and heuristic algorithms [87, 80].

The problem of splitting a large irregular graphs into k parts is a universally employed technique on unstructured grids for finite element, finite difference and finite volume techniques as well as in parallelisation of neural networks simulations, particle calculation and VLSI circuit design. Specifically, in parallel computing, an important application is the mapping of data and/or tasks to processing elements (PEs), with the goals of balancing the load and minimising communication through evenly balancing the weights of the graph parts.

There are several variations of graph partitioning heuristics, naturally, there is a trade-off between runtime and solution quality [88] and, in such cases, the runtime of hypergraph partitioning algorithms is much higher than the one of graph partitioning [84].

The graph partition problem is defined on a graph, $G=(V,E)$, with a set of nodes V and a set of edges E, where, for a weighted graph, each edge $e \in E$ has a weight w_e associated with it. The problem is to find, among all partitions of V into equally sized sets V_1 and V_2, the partition that minimises the total weight of the edges in the cut separating V_1 from V_2.

More generally, the goal is to divide V into equal sized k parts (V_1,...,V_k) while minimising the edges cuts. The graph partitioning problem is already NP-complete for the case $k=2$, which is also called the *Minimum Bisection* problem. If k is not constant the problem is much harder [6]. This partition decision problem is also known as *uniform graph partitioning* or *balanced graph partitioning* and it was proved NP-complete in [49].

The graph partitioning techniques can be mainly categorised as local and global algorithms. The algorithms based on local improvement operate over sub-graphs from an initial partition and try to decrease the *cut size* (sum of weights of edges crossing between subsets) by some local search method. Thus to solve the partitioning problem such algorithms must be combined with some method that creates a good initial partition [45]. Within this category we can mention Kerninghan-Lin [63] and Fiduccia-Mattheyses [42] algorithms.

On the other hand, the global algorithms directly partition a graph into k partitions based on properties of the entire graph. Within this category we can mention spectral partitioning algorithms [43], which the resulting partitions are derived from the spectrum of the adjacency matrix. In this way, the multilevel paradigm is a particular partitioning schema for reducing the computational complexity, and produces high quality partitions in small amount of time [61]. A multilevel partitioning algorithm has one or more stages, basically coarsening, initial partitioning and uncoarsening. In these algorithms, a graph G is first coarsened down to a few vertices, reducing its size, then k partitions of this much smaller graph are computed, and then these partitions are refined to the original graph (finer graph). In such refinements the edge-cut is decreased [61].

3.2 Methodology Overview

In this section, the main characteristics of the proposed load balancing methodology for HPC ABMS applications are described. The objective of the proposed methodology is to provide dynamic workload adjustments in order to effectively enhance the performance of the HPC agent-based application at runtime.

Since there are different ways to model and implement agent-based systems, in our solution we consider spatially-explicit models, which are commonly utilised to model real-world spatial data for studying complex spatial systems [20, 92, 64, 104, 103]. In the literature, spatially-explicit models are also known as agents modelled in "knowledge space" [51], whereby the environment could be a space and the agents have coordinates to indicate their spatial location and define neighbouring communications.

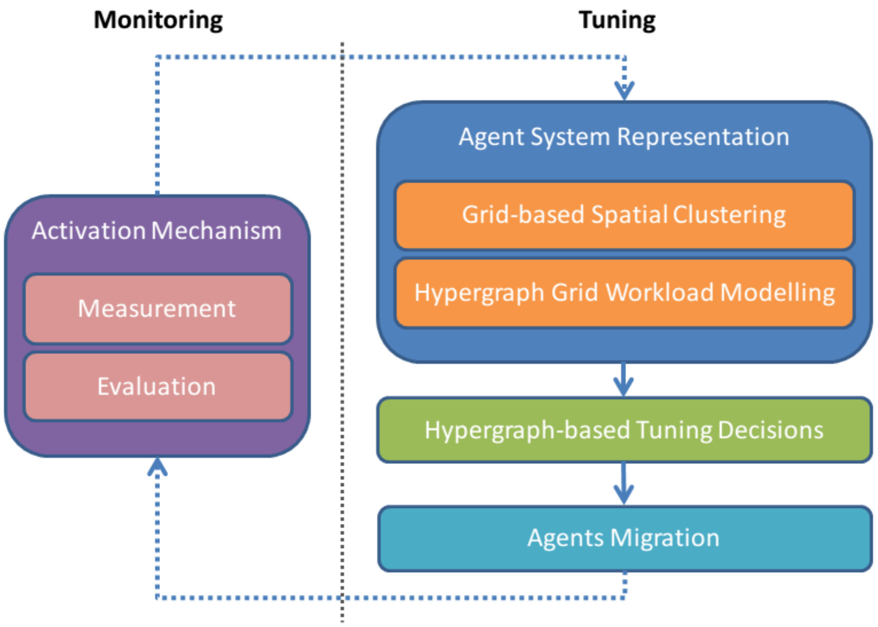
Monitoring
Tuning
Activation Mechanism
Measurement
Evaluation
Agent System Representation
Grid-based Spatial Clustering
Hypergraph Grid Workload Modelling
Hypergraph-based Tuning Decisions
Agents Migration

Figure 3.4: General description of the methodology.

Figure 3.4 shows an schematical representation of this methodology, which consists of two main phases: *monitoring* and *tuning*. In the monitoring phase, the *activation mechanism* measures the parallel application workload at runtime to identify performance problems and evaluates its impact according to a predefined *threshold* value, and, when necessary, it fires the load balancing strategy. The load balancing strategy is defined in the tuning phase, which contains three main stages: the *agent system representation* (ASR), the *tuning decisions* and the *agent migration*. The ASR is carried out through a clustering algorithm based on a grid spatial division (representing groups of agents as annotated grids), then, the global workload is modelled by representing the grids' workload as a hypergraph. Thereafter, the tuning decisions are determined through a hypergraph partitioning algorithm which calculates a new grid configuration. From there, the agent migration is undertaken in order to fulfil the grids reconfiguration decisions (the hypergraph resulting new partitions), and therefore adjusting the global workload.

It is noteworthy that other options to implement ABMS exist, there could be non-spatial representation at all, but agents linked together into a network in which the only indication of an agent relationship to other agents is the list of the agents to which it is connected by network links [100, 50]. In such cases, we do not rule out the possibility of utilising our solution, but the way of building the ASR has to be suited for applying a different clustering algorithm and agents' relations interpreting procedure. For the clustering algorithm, a different relation function or sort criterion has to be defined in order to understand the agents connection characteristics for grouping agents. Once this is done, it is possible to achieve an ASR and, from this, build a hypergraph representation thereafter.

3.2.1 Activation Mechanism

In order to do the appropriate performance tuning, it is necessary to have a thorough knowledge of the application and its agent's development. However, estimation of performance of SPMD ABMS applications is a difficult task. In many instances, the performance varies at runtime by issues such as: amount of computation, interaction pattern between agents, and environmental influences. All these issues may be difficult to analyse and very costly to track in each iteration. Moreover, the resulting performance monitoring and evaluation may frequently activate the tuning phase which is even more costly.

In order to minimise its overhead, the *measurement* stage performs a coarse grain performance tracking, that is to say, only capturing the iteration execution time for all PEs. In the same manner, the *evaluation* stage decides the global workload performance grounded in a permissive *imbalance threshold* value, ensuring the tuning phase is only triggered if really necessary.

The measurement and evaluation stages are detailed hereunder, assuming that the simulation is implemented as a parallel iterative application. In all the expressions, the index j refers to the jth-iteration of the simulation and the index i to the ith-processing element.

3.2.1.1 Measurement

In order to be able to calculate the performance expressions, some application parameters must be measured. These measurements are collected for understanding the runtime workload and, provide a global view of the application performance. In performance model related studies, these parameters have been called the measure points [23, 94, 76, 4] and are the following:

***iteration time** ($t_{iter(ij)}$)* have to be measured in order to calculate the mean processing time μ_j, and the total iteration time.This parameter could be obtained by measuring the wall time of the ith processing element (PE) for the jth iteration.

***number of agents** ($n_{agents(ij)}$)* have to be measured in order to estimate the actual workload(Ew_{ij}) including agent creation and elimination in the ith PE during the jth simulation step. This parameter is obtained by measuring the number of agents at the end of the jth iteration in the ith PE.

Our approach considers these measurements enough to analyse the jth total workload and minimise the overhead of the activation mechanism phase. Logically, other measurements (explained later) will be needed for the tuning phase, but they are only going to be taken if the activation mechanism triggers the tuning phase.

3.2.1.2 Evaluation

We have explained that agent-based applications workload is highly variable and, consequently, achieving an ideal load balancing is nearly impossible. In addition, we have also mentioned that heuristic algorithms should be used for balancing the application workload because getting the optimal load balancing is a NP-hard problem. These algorithms find sub-optimal solutions [66], i.e. the workload balance is improved but it is not perfect, in a reasonable time. For these reasons, we have defined an evaluation expression grounded in a permissive and configurable *imbalance threshold* value.

The imbalance threshold, called just *threshold*, is a value between 0 and 1 that represents a percentage (between 0 and 100) of acceptable imbalance degree in respect to the mean time of all PEs (μ_j for jth iteration), μ_j is calculated with equation 3.1.

$$\mu_j = \frac{\sum_{i=1}^{N_{PE}} t_{iter(ij)}}{N_{PE}} \tag{3.1}$$

With the purpose of detecting imbalances, the permitted *tolerance* time is calculated using the threshold value. With equation 3.2, it is possible to calculate the tolerance time which the permitted time deviation from the mean time and subsequently the *tolerance range* is calculated with equation 3.4 which determines the permitted imbalance bounds in terms of times.

$$tolerance_j = \mu_j \times threshold \tag{3.2}$$

$$tolerance\_range_j = [\mu_j - tolerance_j, \mu_j + tolerance_j] \tag{3.3}$$

Figure 3.5 depicts an example of the iteration runtime tolerance range for 16 PEs, green bars are inside the tolerance range, while orange and blue bars have runtimes out of permitted range.

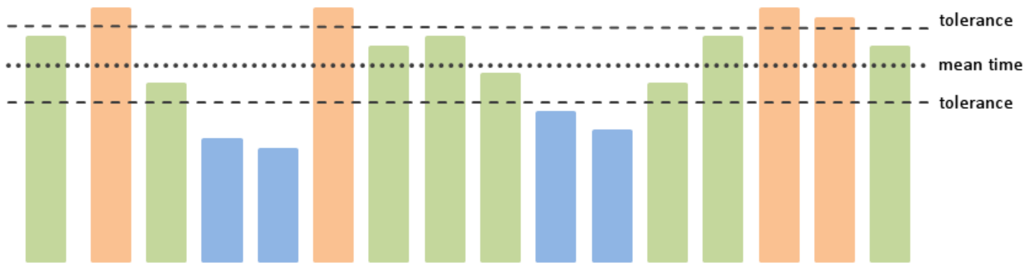

Figure 3.5: PE iteration workloads under tolerance analysis.

The evaluation stage activates the tuning phase when an PE iteration runtime is detected outside the *tolerance_range*, expressed mathematically in 3.4.

$$tuning \implies \forall t_{iter(ij)} \in t_{iter(j)}, \exists t_{iter(ij)} : t_{iter(ij)} \notin tolerance\_range_j \tag{3.4}$$

3.2.2 Agent System Representation

The tuning phase starts with the Agent System Representation (ASR) creation stage. The ASR provide information to diagnose the workload imbalances and help to identify the relation between computing and communication volumes with the agents' behaviours and distribution.

The proposed ASR is comprised of the *Grid-based Spatial Clustering* and the *Hypergraph Grid Workload Modelling* stages. In the Grid-based Spatial Clustering stage, the agents' locations and workload is mapped/traced through a grid-based clustering algorithm. During the Hypergraph Grid Workload Modelling stage, the resulting grids are interpreted and modelled as a hypergraph.

By means of this ASR composition, we are able to provide detailed information of the agent behaviours and PEs performance for dynamic ABMS in such a way to minimise the CPU/memory resource requirements and allow to model indefinitely large domains. The ASR creation proposed stages are detailed below.

3.2.2.1 Grid-based Spatial Clustering

This stage organises the agent locations in relation to the PEs with the application workload. Since agents are continuously changing in terms of quantity, location and workload, the method to associate agent locations, workloads and PEs has to be efficient. In term of resources, the CPU/memory requirements for processing and storing such information may also become unmanageable when a large number of agents with complex interaction rules are simulated.

Our ASR is based on clustering the spatial regions during the simulation into *grids*. First of all, we define a *grid* as a virtual rectangular/cubic region in the 2D/3D space domain. Additionally, each grid covers a unique region in the space. The grid covering extension is established according to a constant named *grid_size*. This *grid_size* is used for defining the *grid_dimensions* which are calculated as $grid\_size^{N_{dim}}$ (N_{dim} is number of dimensions).

The value of *grid_size* should be estimated in accordance with the *influence_-range* of the agents. This influence range is usually called *halo* in the literature [82] and it can be roughly defined as the maximum distance an agent message can reach (it is explained in next subsection). Consequently, grid_size should be also estimated finding a compromise between minimising the number of neighbours of each grid (grid_size=halo), which minimises communication, and minimising the number of grids in order to obtain a manageable the set of grids and reduce the strategy overhead. Basically, if the halo is large then grid_size should be a fraction of the halo, while if the halo is small then grid_size should be a multiple of the halo.

In the same manner, we propose creating grids only in the space occupied by agents in order to reduce CPU/memory requirements, our approach idea is taken from the known technique named *rasterisation*. The defined clustering algorithm is rooted on this principle, so the grids are only created according to the space occupied by agents [85]. Similar approaches have been designed in 2D particle simulations [11]. It is worth mentioning that, thanks to this approach, it is possible to model an indefinitely large domain through only keeping in memory the grid occupied and not storing and analyse the whole space domain.

$$gid_{xyz\cdots N_{dim}} = \begin{cases} gid_x & \leftarrow & \lceil agent_{coordinate\text{-}x}/grid\_size \rceil \\ gid_y & \leftarrow & \lceil agent_{coordinate\text{-}y}/grid\_size \rceil \\ gid_z & \leftarrow & \lceil agent_{coordinate\text{-}z}/grid\_size \rceil \\ & \cdots & \\ gid_{N_{dim}} & \leftarrow & \lceil agent_{coordinate\text{-}N_{dim}}/grid\_size \rceil \end{cases} \tag{3.5}$$

Under our approach, the grids are identified by a unique compound id (*grid_identifier*). The grid_identifier (*gid*) contains one integer value for each dimension of the agents' coordinates, for instance in a 3D model the grid_identifier will be a ternary grid_identifier (a combination of *x,y,z* or $gid_{x,y,z}$). Through equation 3.5 every agent is associated to a unique grid. This equation divides each component of the agent coordinates by *grid_size* and calculate its least succeeding integer, also known as *ceiling* function ($ceiling(x) = \lceil x \rceil$ is the least integer greater than or equal to x). Figure 3.6(a) shows a 2D example of our grid definition.

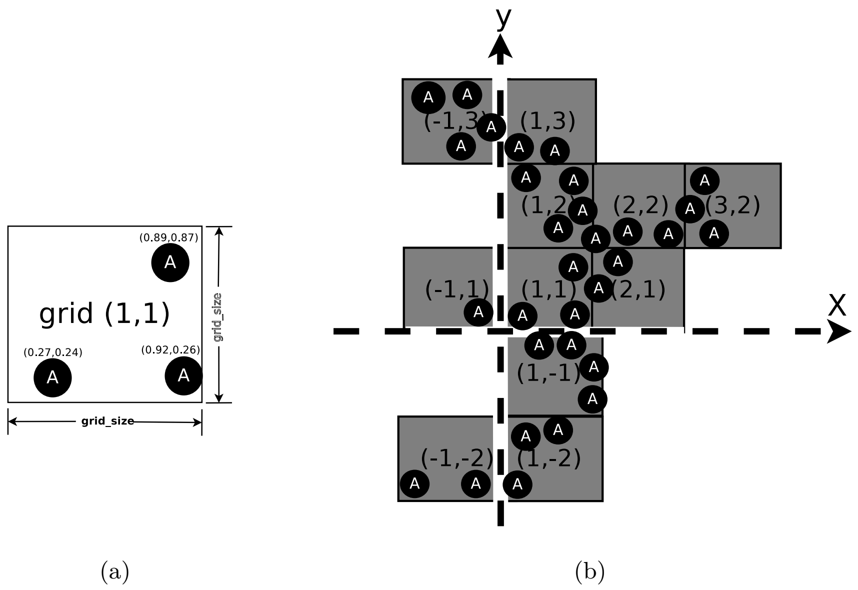

Figure 3.6: (a) Agents contained in the 2D grid with identifier (1,1).
(b) Example of known space in a 2D space.

Using this grid definition, an agent can be associated to a unique virtual grid_identifier with independence of the PEs where it is placed. The virtual space covered by grids is named *known space*, so in case a new agent is located outside of the known space a new grid_identifier appears in our ASR, hence the known space will consist of a set of grids covering only the agents' occupied regions of the space (Figure 3.6(b) depicts this principle in a 2D space).

3.2.2.2 Hypergraph Grid Workload Modelling

As a result of clustering an ABMS domain with the previous grid method, sets of agents are associated to a unique virtual grid. In this stage, we propose to group the exchanges and workload measurements of the agents belonging to each grid in order to analyse and solve the load balancing problem in terms of groups of agents (grids). In this way, it is possible to reduce the computational complexity and the CPU/memory requirements. Additionally, the joint measurements and exchanges of the agents contained in a grid can be modelled using graph representation, more precisely in a hypergraph which allows more accurately modelling the agents system interactions.

As has been mentioned above, the agent influence_range is also known as *halo*. We consider the halo as the spatial region wherein an agent interacts with others outside their own local grid (called grid *influence_zone*). In accordance with the aforementioned, the grid influence_zone is calculated as the union of its contained agents' halos (agents' influence_range), but, depending on the AB model, this may be difficult and costly to calculate and model.

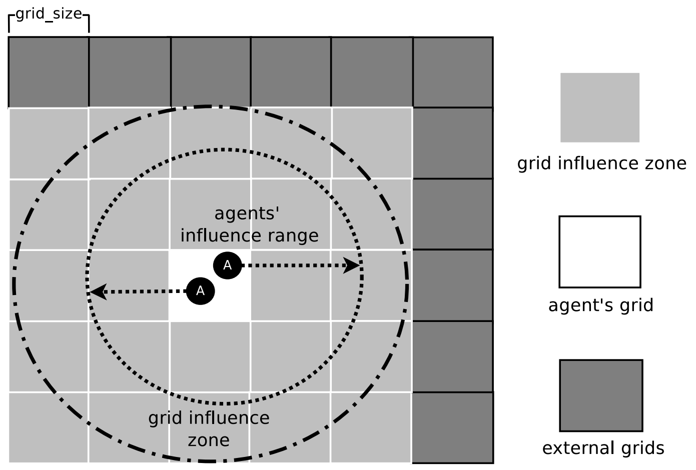

Figure 3.7: 2D example of grid interactions.

In order to simplify the identification of the agents' influence_range operations, we assume a single influence_zone for all the agents of a grid. Figure 3.7 shows this approach. The grid influence_zone is compound by all grids in a distance less than or equal to the distance of its maximum agent halo from its grid bounding box.

$$N_{grid} \leftarrow \left\lceil \frac{max(agent\_range \in gid_{xyz \cdots N_{dim}})}{grid\_size} \right\rceil \tag{3.6}$$

In this approach, it is possible to determine in advance the grids included in the influence_zone using the agent related grid_identifier. First, the number of grids corresponding to an agent influence_range (N_{grid}) has to be estimated with expression 3.6. Then, using 3.7, the grid_identifiers involved in the exchanges can be determined for each grid and, with it, the method is abstracting and simplifying the communication pattern between agents.

$$influence\_zone \implies gid_{xyz \cdots N_{dim}}, \begin{cases} x \in & [x \pm N_{grid}] \\ y \in & [y \pm N_{grid}] \\ z \in & [z \pm N_{grid}] \\ & \cdots \\ \\ N_{dim} \in & [N_{dim} \pm N_{grid}] \end{cases} \tag{3.7}$$

At this point, sets of agent interactions can be represented as grid interactions wherein each grid has, at least, one outgoing exchange with many receiving grids. In this regard, the communication representation may be modelled as a graph but graph edges do not accurately represent the actual interaction of the agents because agent messages may have more than one recipient. For this reason, the hypergraph partitioning approach represents agent interactions better than the graph approach.

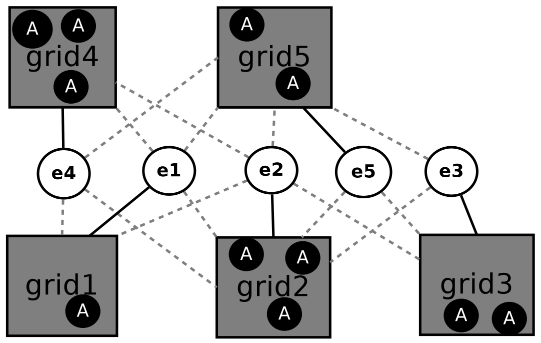

Figure 3.8: Hypergraph modelling example of a 2D system with five grids (vertices $grid1, grid2, grid3, grid4$ and $grid5$) and their outgoing exchanges (hyperedges $e1, e2, e3, e4$ and $e5$).

For this reason, we propose modelling the grids and their exchanges as a hypergraph that represents spatial grids and their interactions as vertices and hyperedges, respectively (see Figure 3.8).

Modelling the simulated system through this hypergraph approach, describes agents' relationships in terms of groups of agents. In the same way, it is possible to determine the groups of agents that communicate more than others by analysing their hyperedge structure and characteristics (communication volume).

Up to now, we have described the generation of an unweighted hypergraph due to a lack of measurement introduction to our ASR so far. However, it is possible to build a weighted hypergraph including computing and communication workload measurements associated to the set of agents of each grid. In this manner, the ASR can also depict more meaningful information for solving imbalance problems.

For these reason, we include as computing measure points the computing time of the agents associated to each grid would provide a more accurately description of the computing volume of the application. Moreover, we include as communication measures the amount of outgoing messages of the agents related to the grids in order to provide a better representation of communication volume of agent subsystems. Including these weights, a more accurate vision of the simulation workload can be modelled by combining computing and communication measurements and, therefore, be more consistent with the objective of diagnosing the workload imbalances and helping to identify the relation between such computing and communication volumes with the agents' behaviours and distribution.

3.2.3 Hypergraph-based Load Balancing

In general, a potential load balancing (LB) solution should rearrange the workloads of PEs with excess (or insufficiency) of computing/communications caused by excess (or insufficiency) of agents and messages among the PEs. In addition, instinctively when an ABMS application requires computing/communication workload tuning, a solution should be implemented with agent migrations constrained by reducing the external-PE agents dependencies.

In terms of workload, we introduce that this particular tuning problem can be essentially expressed as a problem of splitting large irregular components and their interconnections into k parts, wherein each resulting k-part represents a processing element (PE_i) belonging to the set of all available PEs (N_{PE}) in the simulation. Therefore, this proposal perfectly fits the definition of graph partitioning which evenly splits these large irregular components (vertices), while reducing the interconnection cuts (edges).

Following the same grid-based approach defined in the previous subsection, we propose making tuning decisions in terms of groups of agents, this enhances the local and global application performance.

Our tuning decisions are grounded in a hypergraph partition. The problem is defined on a hypergraph, $H=(V,E)$, with a set of grids V and a set of grid outgoing messages E, where each vertex $v \in V$ has a weight w_v associated with it, such that w_v is its joint agent computing time, and each hyperedge $e \in E$ has a weight w_e associated with it, such that w_e is the sum of its outgoing messages. The proper tuning decisions are made according to divide V into equal weighted N_{PE} parts $(PE_1,...,PE_{N_{PE}})$ while minimising the hyperedges cuts.

In accordance with this principle, we partition the hypergraph modelled by our ASR approach into N_{PE} parts, such partitions provide a more appropriate global grid configuration that distributes the workload taking into account computing and communication volumes of the agents. However, partitioning a hypergraph with such balance constraints is complicated, costly, and indeed known to be NP-hard.

Fortunately, there are hypergraph partitioning algorithms capable to operate in parallel platforms. In this regard, there are several approaches of hypergraph partitioning and, of course, there is a trade-off between runtime and solution quality. This indicates that, depending on the expected quality solution, finding a proper grid reconfiguration may take longer than expected. Therefore, to determine an adequate solution quality should be consider, among other things, the problem size, hypergraph complexity, number of PEs and partitioning algorithm efficiency.

Hereinafter, we propose partitioning the hypergraph utilising an existing hypergraph partitioning library in order to efficiently assist a proper workload distribution. In this manner, it is possible to adjust the overhead introduced by such decisions in order to build an efficient LB strategy.

3.2.4 Grid-based Agent Migration

The agent migration is basically the stage that implements the decisions taken in the previous stage by transferring agents from a processing element (PE) to another for modifying the global workload of the parallel simulation. This transference must remove agents from the contributor PE (*sender*) and append them to the recipient PE (*receiver*).

The migration operation can be classified into two procedures: *contribution* and *acquisition*. On one hand, contribution procedure consists of disassociating the agents from their source and sending then to the destination. On the other hand, the acquisition procedure consists of reconnecting the received agents to their respective recipient.

Since the hypergraph-based tuning decisions identify the workload reconfigurations in terms of grids and not in terms of agents, our approach also commands to migrate grids. However, transferring grid by grid to their respective recipient PE is very expensive and inefficient in the current HPC systems.

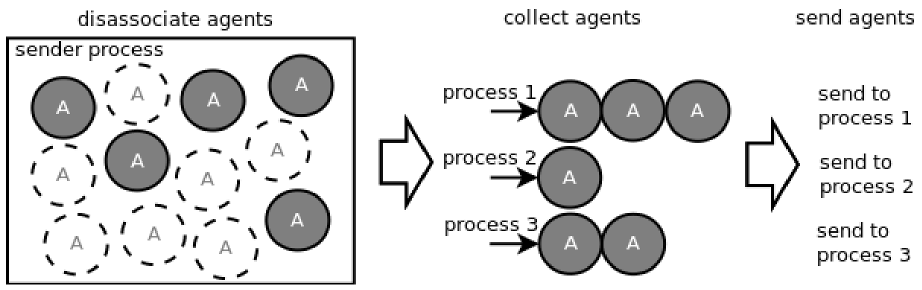

Figure 3.9: Contribution procedure diagram.

For this reason, we have designed a mechanism for migrating agents in a such a way to minimise the number of the required transferences. One approach would be, during the contribution procedure, that each sender emits just one transference for each recipient by collecting all the involved agents before transferring them (Figures 4.7 and 4.8 show this schema). In this way, it is possible to minimise the overhead and, hence, reduce the migration operation impacts over the runtime.

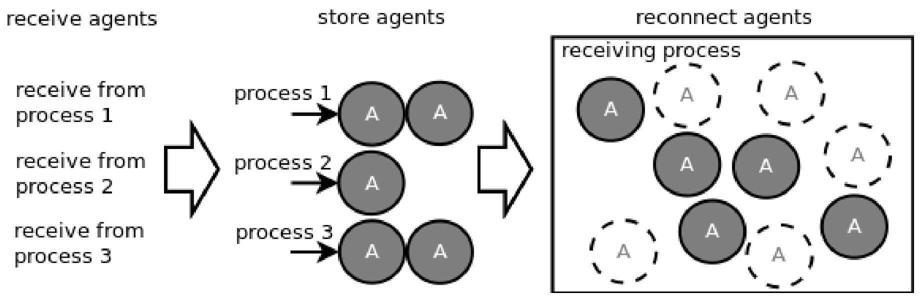

Figure 3.10: Acquisition procedure diagram.

Moreover, the migration operation may also benefit from asynchronous communication and overlapping techniques during the contribution and acquisition procedures. In this manner, overlapping communication and computation, the communication latency can be hidden [68], the migration performance is improved and, consequently, the method overhead reduced.

3.2.5 Agent Messages Management

This point is not a part of the tuning phase in itself, but discusses the significant implication of the message management mechanism in the platform performance and its potential benefit by using our grid-based ASR composition.

First of all, HPC AB simulations involve a large number of agents which are continually communicating with other agents, probably in different PEs, in each simulation step. The large number of agents produces an enormous amount of messages and introduces unfit synchronisation phases which degrades the global performance of the application. In this regard, along with computing workload imbalance, communication bottleneck is the main problem affecting the efficiency of AB simulations. Therefore, it is very important to provide a proper communication management in order to efficiently handle the agent communications.

The efficiency of the platform communication management can be clearly enhanced by sending messages only to the PEs that contain the recipient agents of those messages, avoiding unnecessary collective communications (as opposed to PE-to-PE) which imply a synchronisation point among PEs, that is, all involved PEs must reach a point before they can all begin computing again. These issues are also a difficult task because they require a global view of the distribution of the agents among the PEs.

In spite of that, we propose that our grid-based ASR approach can assist to determine the recipient PEs related to the agents' outgoing messages across all PEs. Through the combination of the *grid_identifier* and *influence_zone* expressions, 3.5 and 3.6-3.7 respectively, it is possible to classify the reach of the outgoing messages in each communication phase in order to send messages only to the location of the recipient agents.

Indeed, taking the benefit of our grid-based spatial organisation, different agent message management schemas could decide which messages need to be keep as internal messages and which need to be dispatched to external. This suitable management concept was proved useful in [72]. In this manner, the comprehension of the agent location with regard to the PEs is facilitated, and, eventually, favourably impacts the platform communication performance.

3.3 Summary

In this chapter, we have discussed the importance of having detailed information of the agent behaviours across the PEs in order to better diagnose the imbalance problems and, therefore, apply tuning decision according to the specific/actual imbalance spots. However, to develop an efficient strategy to reconfigure the workload, the agents communication relations and computing workloads need to be traced, analysed and understood.

In order to comprehend our solution, we considered convenient to briefly introduce some underlying concepts associated to our solution, such as: clustering, domain trimming, graphs & hypergraphs and graph/hypergraph partitioning. According to these ideas/notions, we introduce a load balancing solution that tunes the global simulation workload by migrating groups of agents, simplifies the agent system representation (ASR) construction requirements using a mixed clustering-rastering strategy, understands the simulation through a hypergraph-view that brings a more suitable agent system representation than graphs, and, finally, makes tuning decisions based on a hypergraph partitioning algorithm.

Finally, we wish to add some consideration to develop an efficient agent migration mechanism and, additionally, mentioning that using this methodology it is possible to build an efficient message management mechanism which directly take benefit from the ASR definition.

The chapter that follows describes the implementation of our methodology in a real SPMD ABMS platform as a use case.

Chapter 4

Methodology Implementation

Agent-Based Modelling and Simulations (ABMS) platforms benefit from High Performance Computing (HPC) systems when realistic scenarios with many agents and complex interaction rules need to be simulated. Additionally, in some cases the ABMS platforms require computational scalability features, in such scenarios Single Program Multiple Data (SPMD) paradigm is commonly used. SPMD applications execute the same program in all processing elements (PEs), but on a different set of the domain. Due to its scalability and simplicity advantages, SPMD is the dominant programming model for large-scale distributed-memory machines [58].

Agent-based (AB) simulations show significant variations in amount of computing and communication workload, which are likely to appear during the whole simulation. This causes load imbalances that negatively affect simulation performance. In the meantime, the imbalance problems in ABMS platforms constructed under SPMD paradigm also may experience global (or local) problems related to blockades when some PEs may not move beyond the next global (or local) synchronisation point. For this reason, it is very important to keep the workload well-distributed over the PEs in order to reduce the impact on the application performance [97, 12].

In this regard, these applications need to be assisted by dynamic load balancing (DLB) mechanisms for keeping homogeneous parallel workloads as the simulation occurs. Whilst many load balancing (LB) solutions can be found such as: [119, 118, 110, 105]; these have rarely been incorporated to a multi-purpose environment and, usually, the strategies are highly associated to the nature of the agent model. For this reason, we have described, in previous the chapter, a generalised approach that enables dynamic performance enhancements for HPC agent-based applications wherein agents are spatially-explicit defined and communication dependencies can be determined according to a spatial distance function.

In general, few ABMS general frameworks intended to run parallel simulations on HPC environments can be currently found, as described in [95], RepastHPC [31], D-Mason [32], Pandora [96] and Flame [28] provide a native support of, among other features, parallel executions (see 1.4, Related Studies). However, most ABMS platforms do not provide LB mechanisms. In Addition, commonly LB studies in literature take place in non-SPMD platforms, and most of them use applications created with integrated strategies. Our methodology is tested in the framework Flexible Large-scale Agent Modelling Environment (Flame) which generates SPMD simulation codes for a variety of AB models and, at the time of this research begins, seemed more stable and mature, and was tested in large-scale simulations [29].

Flame allows the automatic production of parallel SPMD code written in C, and MPI for communications. Flame is a template-driven framework intended for large scale AB simulations that does not include any mechanism to balance the workload or routines capable to assist this operation.

Additionally, Flame configures the generated SPMD simulation code for a variety of models utilising rules contained in template files, which can be modified by overriding their rules or adding new template extensions. This fact makes it particularly functional to prove that is feasible to implement our methodology and analyse its performance impact on a real SPMD ABMS platform.

This chapter provides a detailed description of Flame and robustly discusses the modifications and optimisations done to Flame in order to implement and integrate our dynamic load balancing methodology.

4.1 Flame

Flame [28] was developed at the University of Sheffield in collaboration with the Science and Technology Facilities Council (STFC). Flame has been used to solve problems involving multiple domains such as economical, medical, biological and social sciences. This framework facilitates the writing of several agent models using a common simulation environment, and then perform simulations on different parallel architectures, including GPUs [93].

4.1.1 General Overview

Flame is not a simulator in itself, but a tool able to generate the necessary source code for the simulation. It automatically generates the simulation code in C through a template engine using a set of provided template files and the user-provided specification.

The parallel simulation code operates a SPMD paradigm, thereby it implies a unique code replicated among all the computing units and it is composed of

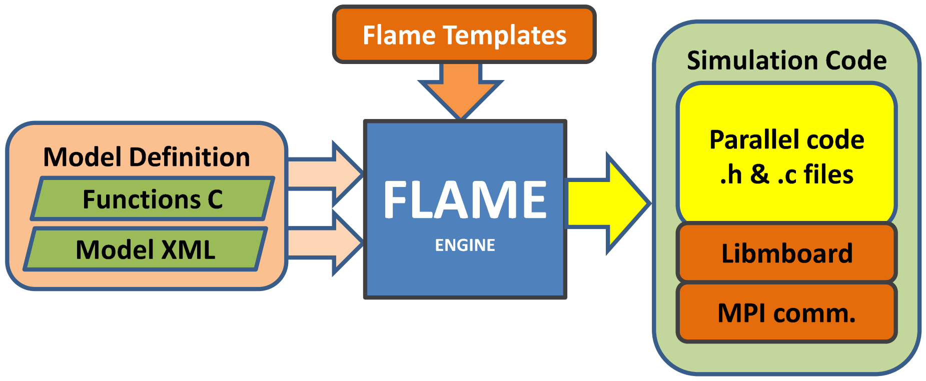

Figure 4.1: Flame basic diagram.

a set of computing and communication phases. The interaction between agents is handled by the message board library *libmboard*, which is implemented in C plus MPI. Figure 4.1 schematically shows the inputs provided to Flame and the output produced by this framework.

The Flame engine uses its template files to get information from the user model specification and generates the simulation code. The template files currently provided by Flame are shown in Table 4.1.

Table 4.1: Flame templates description.

Template	Description
low_primes.tmpl	prime numbers storage.
main.tmpl	main file of the simulator code.
memory.tmpl	agent's routines and structures.
messageboards.tmpl	structures and routines for message boards.
Makefile.tmpl	simulation code Makefile.
partitioning.tmpl	partition methods (R.Robin and Geometric).
rules.tmpl	input filtering rules
timing.tmpl	timing functions.
xml.tmpl	xml reading and writing functions.

The model specification is described by two types of files, XMML (X-Machine Markup Language) files, which is a dialect of XML, and the implementation of the agent functions contained in C files. Codes 4.1 and 4.2 show examples of the XMML definition and C function, respectively, of a simple SIR (susceptible-infected-removed) model.

Code 4.1: XMML definition example of a simple SIR model.

```
</agents>
  <xagent>
    <name>Person</name>
    <description></description>
    <memory>
      <variable><type>int</type><name>id</name><description></description></variable>
      <variable><type>double</type><name>x</name><description></description></variable>
      <variable><type>double</type><name>y</name><description></description></variable>
      ...
    </memory>
    <functions>
        <function><name>move</name><description></description>
          <currentState>1</currentState><nextState>2</nextState>
        </function>
        <function><name>infect</name><description></description>
          <currentState>2</currentState><nextState>3</nextState>
          <inputs><input><messageName>infected</messageName></input></inputs>
        </function>
    </functions>
  </xagent>
</agents>
<messages>
  <message>
    <name>infected</name><description>Position and id of sick agent</description>
    <variables>
      <variable><type>int</type><name>id</name><description></description></variable>
      <variable><type>double</type><name>x</name><description></description></variable>
      <variable><type>double</type><name>y</name><description></description></variable>
    </variables>
  </message>
</messages>
```

Code 4.2: C struct agent attributes example of a simple SIR model.

```
/* Once agent has been sick long enough
   it either recovers and becomes immune or dies. */
int recover() {
    if ( rand()/(RAND_MAX+1.0)*get_sick_count() > LIFESPAN * DURATION / 100.0) {
        if ( rand()/(RAND_MAX+1.0) * 100.0 < CHANCE_RECOVERY ) {
            become_immune();          /* I survived. Yay!! */
        } else {
            GLOBAL_num_agents--;
            return 1;                 /* Oh no!!            */
        }
    }
    return 0;
}
```

This approach is similar to the one followed by Repast and Pandora, but in this case, instead of using an object oriented language, agents state and data are specified using XMML.

4.1.2 Functional Description

The functionality of Flame is based on finite state machines called X-machines, which consists of a finite set of states, transitions between states, messages between agents, and actions (see Figure 4.2). The agent attributes are store in C structures which are defined in the XMML definition files (Code 4.3 shows an example of one these structures).

Code 4.3: C struct agent attributes example of a simple SIR model.

```
/** \struct xmachine_memory_Person
 *  \brief Holds memory of xmachine Person.
 */
struct xmachine_memory_Person {
  int     id;          /**< X-machine memory variable id of type int.           */
  double  x;           /**< X-machine memory variable x of type double.         */
  double  y;           /**< X-machine memory variable y of type double.         */
  double  heading;     /**< X-machine memory variable heading of type double.  */
  int     is_sick;     /**< X-machine memory variable is_sick of type int.      */
  int     is_immune;   /**< X-machine memory variable is_immune of type int.    */
  int     sick_count;  /**< X-machine memory variable sick_count of type int.   */
  int     age;         /**< X-machine memory variable age of type int.          */
};
```

To perform the simulation, Flame holds each agent as an X-machine data structure, whose state is changed via a set of transition functions. Furthermore, transition functions may perform message exchanges between agents.

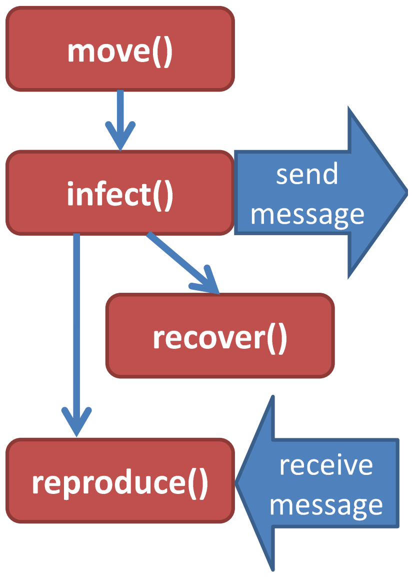

Figure 4.2: X-machine example of a simple SIR model.

The transitions between the states of the agents are accomplished by keeping the X-machines in linked lists. The simulation environment has one linked list for each state of a specific kind of agent. During the simulation, all agents' X-machines are inserted into the list associated to their initial state. Next, the corresponding transition function is applied to each X-machine, and they are moved to the list associated to the agents' next state. This process is repeated until all agents reach the last state, which determines the end of the iteration.

4.1.3 Parallel Functioning

When Flame generates parallel code, this structure is replicated in all the processing elements (PEs) participating in the simulation and the agents are distributed among these PEs. Flame provides two methods of static partitioning for initially distributing agents: *geometric* and *round-robin* partitioning.

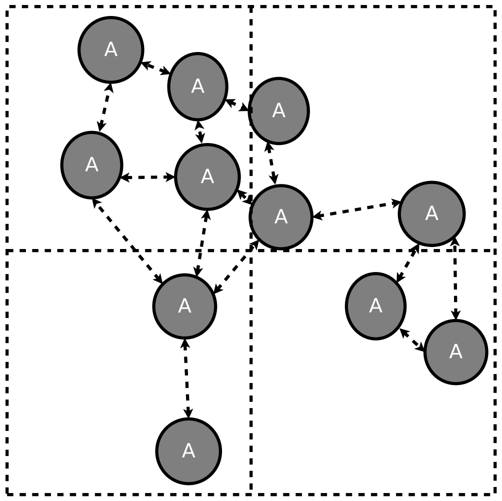

Figure 4.3: Flame geometric partitioning example.

On one hand, the geometric partitioning divides the space into non-overlapping orthogonal regions based on the agent coordinates (as shown in Figure 4.1.3). On the other hand, the round-robin partitioning cyclically assigns agents across all PEs in a manner that each partition contains approximately the same number of agents. In this regard, Flame geometric respects the spatial locality of the agents but usually generates uneven partitions (in term of number of agents). On the other hand, Flame round-robin partitioning generates even partitions without any consideration of spatial locality.

Currently, Flame does not include mechanisms to redistribute agents as the simulation proceeds. Thus, the workload in each PE will rely on the evolution of the model from its initial population of agents. Consequently, the dynamic appearance of imbalance problems cannot be managed.

In addition, a communication library called libmboard, which is built on MPI, is used for managing communication between agents assigned to different PEs. When the Flame engine parses the XMML agent definition generate a set of routines, contained in a C file (*messageboards.(h|c)*), for sending and retrieving messages among agents which are a wrapper to easily access to libmboard operations (Code 4.4 shows an example of a libmboard-provided message routine of a simple SIR model, *add_infected_message*).

Code 4.4: Libmboard-provided routine example of a simple SIR model.

```
/* Post the position of the agent. Should only be called
   for infected agents via condition in XMML. */
int post_position() {
    /* Read agent memory */
    int id = get_id();
    double x = get_x(), y = get_y();

    /* Add message to "infected" board  (id, x, y) */
    add_infected_message(id, x, y);

    return 0;  /* remain alive. 1 = death */
}
```

This library sends all messages to external agents through a coordinated communication mechanism between different MPI processes as shown in Figure 4.4. In this way, Flame provides a general communication mechanism that allows any pair of agents to interchange messages without needing any replication of agents in different PEs.

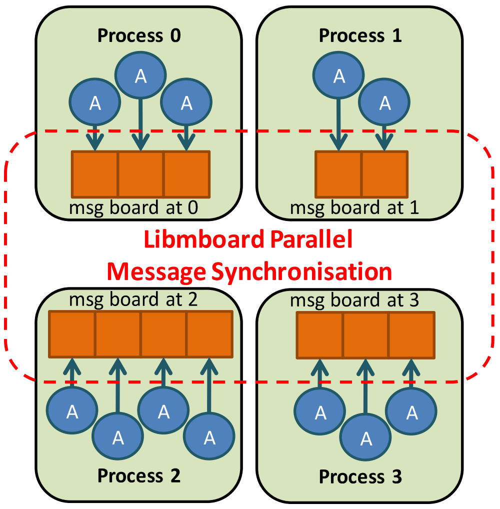

Figure 4.4: Parallel communication and synchronisation via libmboard.

Flame also has some drawbacks, the main one is that its current version does not include any mechanism to enable reorganising agents among the PEs. Thus, the workload in each PE will depend on the evolution of the model from its initial population of agents and there is no load balancing mechanism to mitigate it. In addition, the centralised communication scheme based on libmboard limits the scalability of the generated simulators. These drawbacks do not depend on the particular model being defined and simulated. For this reason, this thesis also includes general proposals for improving the performance of any simulator generated with this framework.

4.2 Hypergraph-based Methodology Implementation

This section describes the implementation of our hypergraph-based methodology through extending Flame with mechanisms for automatically and dynamically balancing the simulator load and for decentralising communication between the PEs participating in the simulation. The implementation suggested below is only for 3D spatially-explicit AB models, however the algorithm can be translated into n-dimensional algorithms by adjusting the number of coordinate axes. As parallel hypergraph partitioner, to provide tuning decisions, we consider integrating the output of a parallel graph tool such as ParMetis [60], PT-Scotch [27], or Zoltan [38].

According to our assumptions, for implementing a load balancing mechanism, it is necessary to be able to monitor, measure, control and minimise the agents computing and communication variations introduced among the PEs. In this way, Flame requires several mechanisms to assist the measurements, performance evaluations, agents migration, tuning decisions, etc. Consequently, Flame has been enhanced for automatically generating efficient routines for assisting these needs. Figure 4.5 illustrates the final block diagram of Flame including all the extensions presented in this thesis. These extensions have been implemented as the following modules: message filtering, migration, measurements, connectivity map, load balancing and graph partitioning, which are explained across this chapter.

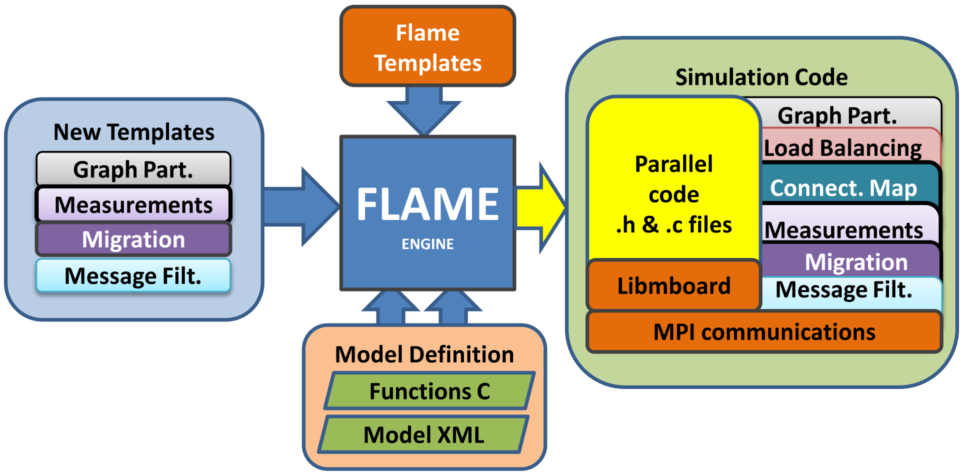

Figure 4.5: Base-diagram of the Flame framework with our extensions.

In order to deliver these new features, we have added new templates to the Flame engine, in such a way, to include in the generated simulation code all the variables, data structures and algorithms to dynamically adjust the workload.

Table 4.2: Flame templates and modifications.

Template	Modifications
main.tmpl	measure points and activation mechanism.
memory.tmpl	libmboard structures initialisation for message filtering.
messageboards.tmpl	definition of message filtering routines.
Makefile.tmpl	new routines and files compilation.

First of all, the Flame template engine has been extended to import our new rules in order to obtain extra information about the agents such as types of agents, the variables (attributes) of the agents, and the size of the agents and their variables. Moreover, rules for obtaining extra measurements related to issues such as platform iteration computing times, messages workload and overheads introduced by Flame and our methodology. Additionally, several routines and enhancements to assist our methodology have been incorporated in the new templates. The template files modified and implemented in Flame are shown in tables 4.2 and 4.3 respectively.

Table 4.3: New templates incorporated.

Template	Description
measures.tmpl	measurement related routines and definitions.
movement.tmpl	agent migration related routines.
mapfiltering.tmpl	ASR related routines and message enhancements.
graphmap.tmpl	hypergraph related routines.

Internally, the template engine uses the new rules contained in our new template files for generating the extra features and enhancements alongside the simulation code. Once the simulation code has been created, the new features and enhancements can be used for performing the global workload reconfiguration.

According to our methodology, for implementing a load balancing mechanism, it is necessary to clearly be able to monitor, measure, control and minimise the agents computing and communication variations introduced among the PEs. Such capabilities were introduced as a mechanism of two phases, *Monitoring* and *Tuning*, and a set of stages: *Activation Mechanism*, *Agent System Representation*, *Hypergraph-based Tuning Decisions* and *Grid-based Agent Migration*. The following subsections describe the implementation of our methodology in Flame along the lines of the phases and stages introduced in the previous chapter. We also consider relevant to provide some control mechanism over the messages sent by the agents in Flame. For these reason, the *Agent Messages Management* is appended before finalising the main stage descriptions.

4.2.1 Activation Mechanism

The performance of SPMD ABMS applications varies during the course of the simulation. Issues such as amount of computation, interaction pattern between agents, and environmental influences are constantly changing depending on the internal state of the agents present in each PEs. For this reason, the implementation of the activation mechanism is executed in each iteration for all PEs. In this manner, it is possible to provide a dynamic evaluation of the workload measurements in every simulation step.

In order to enable measurement operations in Flame, we have added a new template for generating the measurement related routines. The template *measurement.tmpl* incorporates the rules to generate the data structures and algorithms to develop measure points. The measurement routines are specifically generated for each type of agent in the model. The measurement routines are also utilised later for completing the weighted hypergraph.

The following list introduces the main measurement routines and the suffix NAME indicates the name of a specific type of agent. All routines obtain the measurements of the PEs wherein they are called and for a given iteration.

- *get_NAME_msg_size*: gets the size of the messages.
- *get_NAME_num_send*: gets the number of all messages sent.
- *get_NAME_num_send_internal*: gets the number of internal messages sent.
- *get_NAME_num_send_external*: gets the number of external messages sent.
- *get_NAME_time*: gets the computing time.
- *get_NAME_num_agent*: gets the number of agent.

In our approach, the activation mechanism dynamically evaluates the global workload using an imbalance threshold and computing time in each PE. The threshold is a value between 0 and 1 that represents the acceptable imbalance degree. This approach is executed by all the PEs without a central unit of decision. Therefore, each PE must know the global workload situation and execute this stage with the same input. This is done by sharing the iteration computing times between PEs through a collective MPI communication at the end of each iteration. The algorithm 1 describes the activation mechanism.

Algorithm 1 Activation Mechanism

```
collect all computing times for each PE in current iteration
$average\_time \leftarrow \sum computing\_time_i / num\_procs$
$imbalance\_factor_i \leftarrow computing\_time_i / average\_time$
$tolerance \leftarrow threshold * average\_time$
if $\forall i \in procs, \exists\ proc_i / \ |imbalance(proc_i)| \geqslant tolerance$ then
  launch Tuning Phase
end if
```

In this manner, all PEs perform the same workload evaluation and it is possible to make tuning decisions before starting the next iteration. Furthermore, as every PE executes this evaluation with the same inputs (times and agents of all PEs), every PE obtains the same balanced and imbalanced PEs. The mechanism is triggered when an imbalance factor is detected outside the tolerance range. The tuning phase is launched if a computing time is detected outside of this tolerance range.

4.2.2 Agent System Representation

Here, the agent system has to be modelled in order to provide a suitable reduced representation that interprets the agents' behaviours and interactions. All PEs build a sub-hypergraph that contains the representation of the agents assigned to the PE. These sub-hypergraphs are passed to the parallel partitioning tool in order to determine the tuning decisions.

The ASR is carried by clustering the agents of a 3D space into groups called *grids* which represent a virtual unique cubic region of the simulated space. This *grid_size* is used for defining cubic regions of dimensions *grid_size* x *grid_size* x *grid_size* ($grid\_size^3$). The value of *grid_size* should be estimated in accordance with the *influence range* of the agents. The grids are identified by a unique *ternary id* which is a combination of the x-, y- and z-axis ids.

Algorithm 2 Grid-based Spatial Clustering

```
$grid\_size \leftarrow$ grid length
$grid\_group_i \leftarrow$ known space in $PE_i$
for all $agent \in PE_i$ do
  $x \leftarrow x$-coordinate of $agent$
  $y \leftarrow y$-coordinate of $agent$
  $z \leftarrow z$-coordinate of $agent$
  $gid_x \leftarrow ceil(x \;/\; grid\_size)$
  $gid_y \leftarrow ceil(y \;/\; grid\_size)$
  $gid_z \leftarrow ceil(z \;/\; grid\_size)$
  $agent\_grid\_id \leftarrow \{gid_x, gid_y, gid_z\}$
  if $agent\_grid\_id \in grid\_group_i$ then
    increase agent counter in $agent\_grid\_id$
  else
    add $agent\_grid\_id$ to $grid\_group_i$
  end if
end for
```

Algorithm 2 shows the *Grid-based Spatial Clustering*. As a result, every agent will be assigned to only one grid, and all grids will have a positive agent counter.

These grids are constructed along with the exploration of the existing agents across the PEs. The space covered by grids is named *known space*, so new grids will appear when an agent is located outside of the known space. In case an agent is located within the known space, the agent counter belonging to its agent's grid region is increased.

In our implementation, the estimation of the retrieving message regions is performed through a Euclidean distance calculation, and not only considers the agent interaction region. Alternatively, it is determined by calculating the maximum message range from the grid boundaries. In this way, agent interaction patterns can be collapsed as representing grid interconnections and subsequently as hyperedges. The *grid_range* is calculated by dividing the agent interaction range (*agent_range* in Algorithm 3) by *grid_size* value, and consequently represents the grid's *halo* (interval $[g_{xyz} - grid\_range, g_{xyz} + grid\_range]$). Algorithm 3 shows the procedure to detect the grid-based interactions in order to assist the *Hypergraph Grid Workload Modelling.*

Once the Grid-based Spatial Clustering and Grid-based Interaction Mapping algorithms are performed in each PE, the simulation environment is capable of determining the range of the messages between agents, and hence understanding communications between PEs. From here, each PEs has a certain simplification of the agents placed in such PE and their interactions in terms of grids. At this point, each grid represents a vertex and its interactions hyperedges of a sub-hypergraph stored in a particular PEs.

4.2.3 Hypergraph-based Tuning Decisions

On this point, the implementation requirements to perform tuning decisions are discussed. The tuning decision implementation are carried out with assistance from the outcome of the parallel hypergraph partitioner called Zoltan [15] which is, to the best of our knowledge, the only well-established partition library supporting the possibility of being used at runtime (Zoltan configuration is explained in the next chapter). In this regard, it is necessary to adapt all the ASR sub-parts (the sub-hypergraphs in each PE) to the functional needs of the parallel graph tools.

For instance, one of these requirements is that the vertices of the sub-hypergraphs must have a unique global identifier represented as an unsigned integer. This requirement makes that the grid configuration of each PE has to be globally known in order to generate unique grid identifiers.

Another requirement is that the vertices of the sub-hypergraph have to only exist in one PE. Since Flame generates parallel ABMS code with dynamic creation/deletion of agents and the agents are constantly moving in the space, new agents may appear in any place of the space or existing agent may move across the grids, or die potentially leaving empty grids.

Therefore, the platform requires implementing the dynamic creation and

Algorithm 3 Grid-based Interaction Mapping

```
$grid\_size \leftarrow$ grid length
$grid\_range \leftarrow ceil(agent\_range/grid\_size)$
$grid\_group_i \leftarrow$ known space in $PEs_i$
$global\_group \leftarrow$ known space across all PEs
**for all** $agent \in PEs_i$ **do**
  $x \leftarrow x$-coordinate of $agent$
  $y \leftarrow y$-coordinate of $agent$
  $z \leftarrow z$-coordinate of $agent$
  $gid_x \leftarrow ceil(x\ /\ grid\_size)$
  $gid_y \leftarrow ceil(y\ /\ grid\_size)$
  $gid_z \leftarrow ceil(z\ /\ grid\_size)$
  $agent\_grid\_id \leftarrow \{gid_x, gid_y, gid_z\}$
  **for all** $grid\_id \in global\_group$ **do**
    $g_x \leftarrow x$-component of $grid\_id$
    $g_y \leftarrow y$-component of $grid\_id$
    $g_z \leftarrow z$-component of $grid\_id$
    **if** $gid_x \in [g_x \pm grid\_range] \wedge$
   $gid_y \in [g_y \pm grid\_range] \wedge$
   $gid_z \in [g_z \pm grid\_range]$
 **then**
      $grid\_id$ is within the interaction region
    **else**
      $grid\_id$ is without the interaction region
    **end if**
  **end for**
**end for**
```

reconfiguration of the hypergraph. If this is not controlled, grids and their identifiers may be duplicated in different PEs but containing different agents. In this regard, an agent migration is performed before creating the sub-hypergraph, in so doing, all the agents associated to a grid are located in only one PE.

For assisting the hypergraph creation and incorporating a parallel hypergraph partitioner, we have implemented a template. The template *graphmap.tmpl* , in conjunction with the Flame engine, enables the automatic generation of the of the routines described below. The following list introduces the main routines.

- *create_map3D*: builds an adjacency matrix based in our ASR.
- *graph_partitioning*: builds a hypergraph from the resulting adjacency matrix.
- *get_exchange_map_data*: returns the adjacency matrix.
- *push_vertex*: creates a vertex.

- *find_vertex*: returns a vertex for a given grid id.
- *push_edge*: creates a hyperedge for a given vertex.
- *fill_pins*: creates hyperedges from the adjacency matrix.

As the tuning decisions are taken from the outcome of parallel graph tools, the adjacency structure of the sub-hypergraphs are stored using the compressed storage format (CSR), which is a widely used scheme for storing sparse graphs [60]. Each sub-hypergraph, stored as a sparse matrix, represents a part of the global hypergraph that models the entire agents' system.

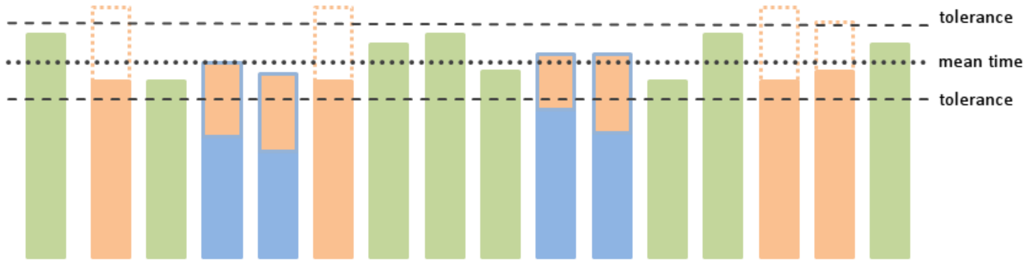

Figure 4.6: Example of PE iteration workload within the tolerance range.

Later, each PE calls the parallel hypergraph partitioner. On the one hand, the parallel hypergraph partitioner, for each PE, returns the vertex ids that should be exported to other sub-hypergraphs and, on the other, the vertex ids that should be imported from other sub-hypergraphs. Therefore, since the hypergraph models the ASR and each sub-hypergraph represents the part of the ASR that belongs to a particular PE, the import/export outcomes are extrapolated to an operation of multiple grid transferences between PEs. In this regard, the agent migration across the simulation PEs is commanded in terms of grids.

Further on, after the agent migration occurs, the simulation is resumed and continues with a new workload configuration that satisfies the tolerance range. Figure 4.6 shows an example of workload reconfiguration wherein bars represent PE workload of an iteration and dashed blocks represent transferred workloads (from orange bars to blue bars).

4.2.4 Grid-based Agent Migration

In order to accomplish any sort of workload reconfiguration the ABMS platform needs to enable efficient mechanisms for transferring agents between PEs.

Consequently, Flame has been enhanced for automatically generating efficient routines for migrating agents. In order to deliver this new feature, we have added a new template for generating the migration routines. It is *migration.tmpl*, which generates variables, data structures and algorithms to carry on the migration process. The template engine has been modified to process this template to obtain the information about the agent types, the variables (attributes) of the agents, and the size of each agent variable.

Once the simulation code has been created, the migration routines can be used for moving agents between simulation PEs. The migration process can be subdivided into two procedures: *contributing agents* and *acquiring agents*. Algorithms 4 and 5 show the procedures involved during the migration of agents.

Algorithm 4 Contributing Agents

```
while agents to be sent do
  insert in the recipient list
end while
calculate the sizes and create the buffers
pack the agents and send the packages
```

Algorithm 5 Acquiring Agents

```
create the memory buffers and receive the agents
while packed agents do
  unpack and insert agent in the current PE
end while
```

The *contributing* procedure consists of removing, packing (serialising) and sending the selected agents in the sender PEs. This procedure holds a migration list for each target PE and type of agent. Then, the agents to be migrated are extracted from the simulation PE X-machine list associated to its current state and inserted in the corresponding migration list. Once all migrating agents have been inserted in the appropriated list, they are serialised in a set of contiguous memory buffers to be packed, using the corresponding MPI functions, in order to be sent in a single message to a specific receiving PE. Finally, the message is asynchronously sent to the receiving PE for overlapping the creation of the next message with the communication of the previous one.

The *acquiring* procedure consists of receiving, unpacking (de-serialise) and adding the agents in the recipient PEs. The messages with packages of agents arrive to the recipient PE in buffers that must be unpacked using the corresponding MPI functions. Once the agents X-machines have been unpacked, they are inserted in the list associated with their state alongside the other agents in the recipient PE.

The migration routines are specifically generated for each type of agent in the model, and it is possible to perform migrations after any transition. The following list introduces the main migration routines. The suffix NAME indicates the name of a specific type of agent.

- *Pop_NAME*: moves agents X-machines to a specific linked list and removes them from the current PE.
- *Pack_NAME*: packs (serialises) all agents X-machines kept in the linked lists in contiguous memory buffers, one buffer for each recipient.
- *Send_NAME/Recv_NAME*: prototypes to define how to send and receive sets of packet agents.
- *Unpack_NAME*: unpacks (de-serialise) agents X-machines from the buffer to the appropriated object.
- *Push_NAME*: adds an agent to the current PE inserting the received agent into the adequate X-machine list.

As the tuning decisions responds to a grid-based ASR, its proposed reconfiguration outcomes are also returned in terms of grids. Such outcome is composed of two lists for each PE, a list of grids to be imported from other PEs and the list of grids to be exported to other PEs.

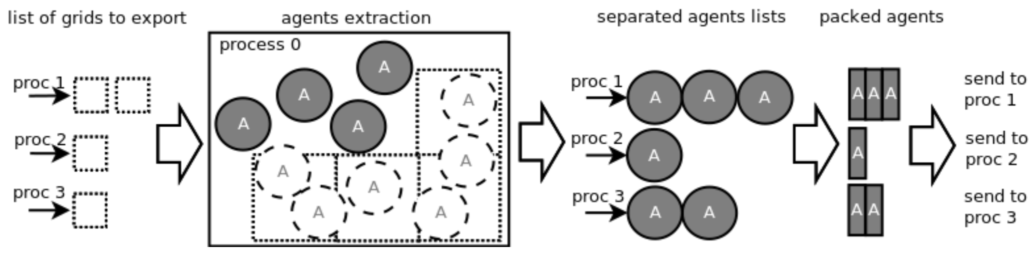

Figure 4.7: Contribution procedure implementation.

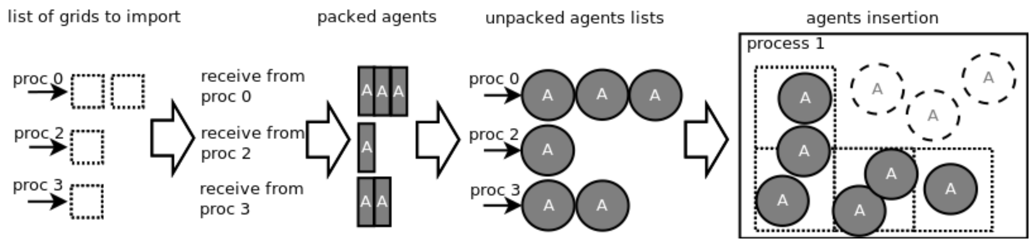

Figure 4.8: Acquisition procedure implementation.

This fact led the agent migration stage to firstly identify all the agents associated to such grids. Once this has been done, each PE extracts, packs and sends agents associated to the list of grids to export and receives, unpacks and inserts agent associated to the list of grids to import. Figures 4.7 and 4.8 depict the implementation of such procedures.

4.2.5 Agent Messages Management

The efficiency of the platform communication management is a very significant issue in order to enhance the application performance. The parallel applications may improve the global performance by avoiding unnecessary collective communications. In this way, providing a decentralised communication mechanism by enabling PE-to-PE communications may really help.

This issue is also a difficult task in HPC ABMS applications because it requires a global view of the distribution of the agents among the PEs and, moreover, the simulations involve a large number of agents which are continually communicating.

The communication mechanism of Flame provides a lower abstraction level mechanism that allows any pair of agents to interchange messages without needing any PE reference. Flame uses broadcast for implementing its board approach, leaving the recipient agents the possibility of using input filters for choosing which messages to read. This communication management does not efficiently handle the agent communications because any agent message exchange involves all PEs by stopping them to complete the all-to-all broadcast.

However, Flame does not publicise routines intended to send messages to specific PEs, although its communication library *libmboard* is able to do this task through the MB Filter*-family functions.

These functions facilitate targeting of a set of specific PEs for sending the data of local message boards (outgoing messages). For doing so, a *filtering_function* must be provided by the user. This filter must receive two arguments: a pointer to the message and the rank of an MPI process, and it must return 1 if the message has to be sent to the PE with the given rank or 0 otherwise.

Code 4.5: Board and filter initialisation

```
/* Create an MB Board object */
 MBt_Board board_msg;
 MB_Create(&board_msg, sizeof(msg));
/* Create an MB Filter object */
 MB_Filter filterG;
/* Link Code-2 to the filter */
 MB_Filter_Create(&filterG, &isTargetPid);
/* Assign that filter to the board */
 MB_Filter_Assign(board_msg, filterG);
```

Through Code 4.5, the *MBt_Board* and the *MB_Filter* objects are created, later on, the filtering function (Code 4.6) is assigned to the *MB_Board*. Code 4.6 represents the filtering function, which is written in accordance with the previously defined *Grid-based Interaction Mapping* stage (Algorithm 3).

Code 4.6: Filtering function

```
int isTargetPid(const void *msg, int pid) {

 /* Extract grid id from a message */
 int gidx, gidy, gidz, GRID_RANGE;
 gidx = ceil(msg->x/GRID_SIZE);
 gidy = ceil(msg->y/GRID_SIZE);
 gidz = ceil(msg->z/GRID_SIZE);
 GRID_RANGE = ceil(agentRange/GRID_SIZE);
 arrayGrids = &gridsInPE[pid];

 /* Determine grids in influence zone */
 int gx, gy, gz;
 for( i = 0 ; i < arrayGrids.size ; i++ ) {
  gx = arrayGrids.grid[i].gx;
  gy = arrayGrids.grid[i].gy;
  gy = arrayGrids.grid[i].gy;

  if( gidx <= gx + GRID_RANGE &&
      gidx >= gx - GRID_RANGE &&
      gidy <= gy + GRID_RANGE &&
      gidy >= gy - GRID_RANGE &&
      gidz <= gz + GRID_RANGE &&
      gidz >= gz - GRID_RANGE )
   return 1;
 }
 return 0;
}
```

Using these filters, each simulation PE can create separate buffers for each remote MPI process, sending only the relevant messages to each PE and avoiding global communications at the cost of creating more buffers and messages. The reason why Flame does not publicise these functions is that using them requires users to know which agents are assigned to each simulation PE, which is clearly in a lower abstraction level with respect to using input filters.

Nevertheless, we have implemented a new template in order to generate output messages filtering routines using the MB Filter* functions. The new template is *mapfiltering.tmpl*, which generates the message filters according to the agent's message phases. The following list introduces the main measurement routines and the suffix MSG indicates the name of a specific type of agent message.

- *add_MSG_message_filtered*: adds an outgoing message to a given recipient PE.
- *add_MSG_message_filtered3D*: adds an outgoing message for automatically filter its recipient PEs.
- *isTargetProc_MSG*: automatically calculates the recipient PEs of a message.

This agent message management implementation directly takes benefit of our grid-based ASR and decentralise the communication schema of Flame. In this way, it is possible to decide which messages need to be kept as an internal PE message and which need to be dispatched to an external PE [72].

4.3 Summary

Throughout this chapter, we have shown how certain modifications and mechanisms need to be integrated in SPMD ABMS applications in order to be benefited by the proposed performance tuning methodology. The methodology is proposed in the previous chapter and considers spatially explicit models with neighbouring communications. The methodology builds an ASR from clustering the application domain in grids which, in conjunction with the agents representative interactions, is modelled as a weighted hypergraph. Then a hypergraph partitioning tool determines a better grids configuration which, finally, implies agent migration.

Moreover, this chapter describes a set of algorithms and mechanisms to integrate our methodology in a real HPC ABMS platform and, in this manner, be able to test the methodology. For achieving this, several modifications and extensions have been implemented in the framework Flame to evaluate the proposed tuning methodology described in Chapter 3.

The framework Flame is extensively described in Section 4.1 which is a tool to generate SPMD code for simulating AB model. Through the user-provided AB model specification and a set of Flame-provided template files, the Flame engine generates such SPMD codes. Additionally, modifying the Flame templates or adding new ones, it is possible to provide enhancements to all generated parallel codes. In this regard, implementing our methodology in this framework provides tuning features to a wide range of SPMD AB simulations. Therefore, attaching our methodology to this tool facilitates the performance observation and analysis of an extensive range of use case models. Lastly, we also described the implementation of an efficient message management mechanism which directly take benefit from the ASR definition and improves the original Flame communication mechanism.

The next chapter presents the evaluation of our methodology using the implementation introduced in this chapter.

Chapter 5

Experimental Results

In this chapter, we present the evaluation of the hypergraph-based methodology to reconfigure the workload of SPMD ABMS applications through migrating agents. Up until this point, we have defined the methodology and described its implementation in a real platform called Flame.

In Chapter 3, we proposed a generalised approach that enables dynamic performance enhancements for SPMD ABMS applications wherein agents are spatially-explicit defined and communication dependencies can be determined according to a spatial distance function. Given the intrinsic dynamic workload variability throughout the simulation execution, the strategy that balances the workload is addressed dynamically. The proposed solution tunes the global simulation workload by migrating groups of agents, simplifies the ASR construction requirements using a mixed clustering-rastering strategy, models the simulation through a hypergraph-view and, finally, makes tuning decisions based on a hypergraph partitioning algorithm.

Chapter 4 describes how the methodology was implemented in the Flame framework, which is capable to generate SPMD code from a user-defined agent-based (AB) model definition. We implemented modifications and mechanisms in Flame in order to include our tuning features in the Flame outcomes, the SPMD ABMS generated codes. In this way, we are able to incorporate the methodology to different use case models.

For these reason, in section 5.1, we firstly introduce three AB models used in the experimental evaluation. These models are implemented in our extended version of Flame and, consequently, their generated codes include our tuning features. In this manner, the performance observation and analysis of SPMD ABMS applications is facilitated.

Later, in section 5.2, we present the experimental validation of the proposed methodology. This section shows the configuration of the experimental environments and the evaluation of the implementation of different enhancements and mechanisms proposed in this thesis.

5.1 AB Use Case Models

This section focuses on introducing the selected AB models for the evaluation of the methodology. These models are Susceptible-Infected-Remove, Colorectal Tumour Growth and Keratinocyte Colony Formation, which develop dynamic changes in terms of workloads grounded in random rates of creation and elimination of agents. These models are briefly described in following subsections 5.1.1, 5.1.2 and 5.1.3.

5.1.1 Susceptible-Infected-Removed Model

The Susceptible-Infected-Removed (SIR) model describes the spread of an epidemic within a population on a 2D toroidal space [37]. The population is divided into three groups: Susceptible (S), Infectious (I), and Recovered (R). For this reason, this model is called SIR. In summary, the susceptible are those individuals who are not infected and not immune, the infectious are those who are infected and can transmit the disease, and the recovered are those who have been infected and are immune. Figure 5.1 shows an graphical example of the agent distribution of this model.

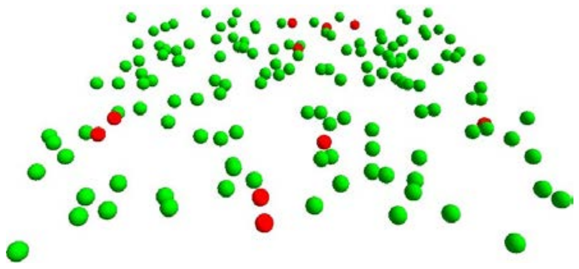

Figure 5.1: Spatial agent distribution example of SIR model, green and red circles represent susceptible and infectious people respectively.

Additionally, natural births and deaths during the epidemic are included in this SIR model, so individuals could die from the disease or by natural death due to ageing. Consequently, births and deaths represent a dynamic creation and elimination of agents. The main states of this model are: get_older, move, infect, recover and reproduce. Due to the random distribution of death and birth rates, the simulation workload changes dynamically and, hence, load imbalances are likely to appear.

5.1.2 Colorectal Tumour Growth Model

The Colorectal Tumour Growth (CTG) model represents an expansive in-vivo tumour development behaviour [7, 59]. The model used in this thesis includes the basic biological properties of tumour cells (TC) such as cell growth, proliferation, and death, located within a vascular network and further considers dependence on nutrients (oxygen). Two tumour growth stages may be considered; the first is defined as a vascular growth when tumours depend on simple diffusion of nutrient supply. The second stage involves vascular growth; *angiogenesis* is a multi-step physiological process which is initiated when tumour cells become increasingly hypoxic and, in response, secrete angiogenic factors such as vascular endothelial growth factor (VEGF). These diffuse and stimulate the existing vessels to form new sprouts, which migrate and connect to other sprouts or to the existing vascular network, forming new blood vessels. This results in an abnormal tumour vasculature which is leaky and tortuous.

The model computes, among other things, oxygen concentration which affects the cell cycle. VEGF concentration is calculated after checking for cell division, death and movement. This induces tumour angiogenesis which initiates vessel growth. Lastly, vessels are checked for collapse. The entire process is repeated for a fixed duration of time.

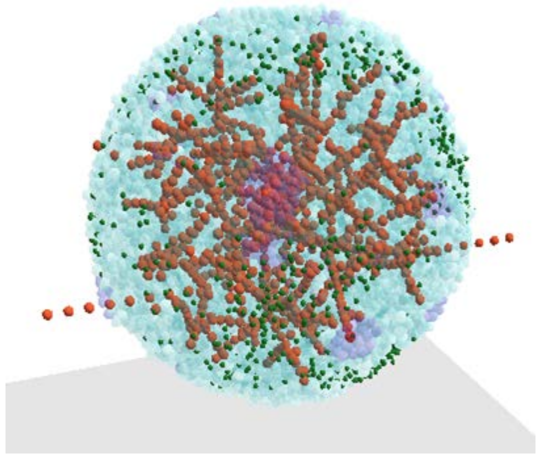

Figure 5.2: Example of Colorectal Tumour Growth.

In terms of implementation, this model is composed of tumour cells (TC), tumour-associated endothelial cells (TEC) and helper agents. Cell agent represents both TCs and TECs. The TC agent consists of a group of TCs, while the TEC agent indicates one TEC. The Helper agent is used to compute the last cell Id and is run only once for each iteration. Figure 5.2 shows a graphical representation of this model. In this figure, TCs which have a smaller oxygen level than the threshold are displayed in cyan, and TCs in proliferation with an oxygen level greater than the threshold in purple. TECs are displayed in red. TECs which have VEGF levels over the threshold, and hence can divide, are shown in yellow.

The predefined rules on memory variables of the agents are defined for updating the position of the agents, cell cycle, oxygen concentration, VEGF concentration and variables related to angiogenesis. The number of functions the agents have is also introduced to perform tumour growth coupled with angiogenesis, such as output_location, resolve_forces, update_rel_c_oxy, update_rel_c_gf, cycle, collapse_TECs and update_last_cell_id_helperagent.

In each iteration, the agents interact computing the expansion coordinates of new cells, forces among the cells, amount of nutrients that comes from the vessels, amount of oxygen permeated through the cells as well as new coordinates using the resulting forces.

Later, these procedures will determine either the growth or the death of each TC or TEC. Consequently, this model presents computing/communication workload imbalances as the simulation proceeds.

5.1.3 Keratinocyte Colony Formation Model

The Keratinocyte Colony Formation (KCF) model was developed, based on rules derived from literature, for predicting the dynamic multicellular morphogenesis of normal human keratinocytes (NHK) and of a keratinocyte cell line ($HaCat$ cells) under varying extracellular Calcium (Ca^{++}) concentrations. The model enables in virtuo exploration of the relative importance of biological rules and was used to test hypotheses in virtuo which were subsequently examined in vitro [107, 108]. The superficial epidermis is largely composed of keratinocytes that are formed by division of cells in the basal layer, and give rise to several distinguishable layers as they move outwards and progressively differentiate (change their types).

The implementation of this model is a simplification of the KCF model published in [107, 108]. This AB model is composed of two parts: the agents, in this case representing cells; and the environment, here being the user-defined flat square surface with 'walls' in which the cells reside, along with global factors such as extracellular calcium. Each cell was modelled as a non-deformable sphere which was capable of migration (slow movement of cells), proliferation and differentiation (changes from one cell type to another).

The KCF model is composed of five kind of agents: stem, transit-amplifying, committer, cornified and HaCaT cells. Initially, agents (cells) output their location and type (stem cell, transit amplifying (TA) cell, committed cell, corneocyte) to the message lists for other cells to read. Each cell then performs rules specific to its own position in the cell cycle. Following this, cells decide whether to change to another cell type, based on the differentiation rules in the model. Cells then execute their migration and physical rules. All rules are executed in the context of the agent's own internal state and its immediate environment as discovered through interrogation of the message lists.

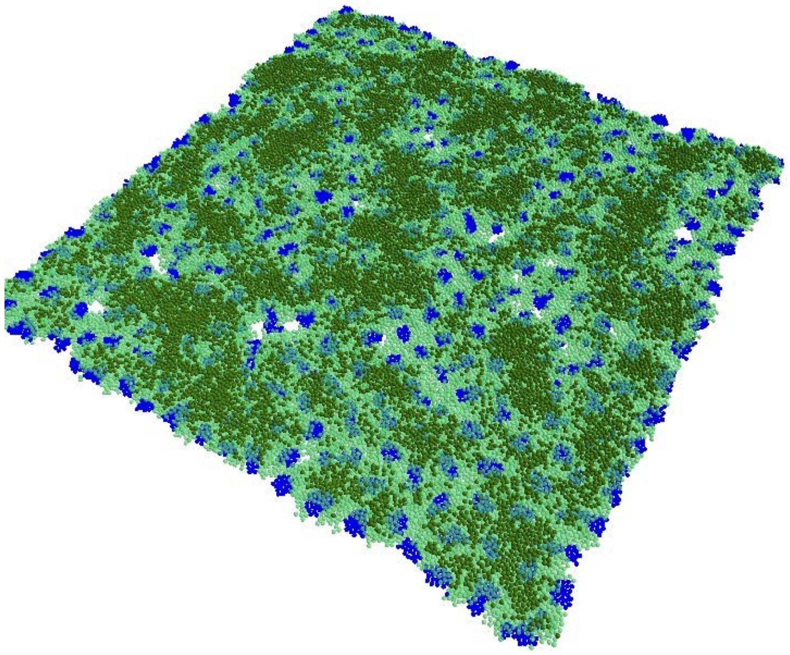

Figure 5.3: Example of Keratinocyte Colony Formation.

Figure 5.3 shows an example of this model. The modelled cells are stem cells (blue), TA cells (light green), committed cells (dark green) and corneocytes (brown). Each of these agents also performs distinguished cell cycles, thus their implementations are composed different states with different computing and communication workload. The cell differentiation along with the cellular proliferation generate serious imbalance problems which also require a dynamically load balancing strategy to readjust the workload during the simulation.

5.2 Experimental Evaluation

This section focuses on introducing the HPC environment, configurations and the evaluation of different aspects of the methodology presented in this thesis. Additionally, this section demonstrates that SPMD ABMS applications require dynamic load balancing mechanisms to improve their performance and make more efficient utilisation of the HPC resources.

Moreover, all experiments show significant gains and improvements in real AB models versus their executions without the methodology proposed in this thesis.

In subsection 5.2.1, the test environments and the clusters configuration, where experiment were executed, are described. Next, the agent migration is analysed and advantages of using the ASR defined in this the thesis in conjunction with message management mechanism is discussed. Later, the methodology including all the performance enhancements working together is shown. The last experiment analyses the overhead associated to the hypergraph partitioning.

5.2.1 Experimental Environments

The main objective of this section is to summarise the environments wherein the experiments were carried out. The experiments were executed on homogeneous linux computing clusters, either on *Cluster IBM* or *Marenostrum III*.

Cluster IBM environment is configured with the following features: 32 IBM x3550 Nodes, 2xDual-Core Intel(R) Xeon(R) CPU 5160 @ 3.00GHz 4MB L2 (2x2), 12 GB Fully Buffered DIMM 667 MHz, Hot-swap SAS Controller 160GB SATA Disk and Integrated dual Gigabit Ethernet. The operating system is SUSE Enterprise 10 SP1 with software setup: Flame 0.16.2/0.17.0, libmboard 0.2.1/libmboard 0.3.1 and OpenMPI 1.4.1.

Marenostrum III contains heterogeneous nodes; our experimental environment is configured with the following IBM dx360 M4 node features: 2x Intel SandyBridge-EP E5-2670/1600 20Mă8-core at 2.6 GHz, 8x4 GB DDR3-1600 DIMMS (2GB/core), Gigabit 10Gbit Ethernet network used by the GPFS Filesystem and Infiniband Mellanox FDR10 for High bandwidth network used by parallel applications communications (MPI). The operating system is SuSe Distribution 11 SP3 with software setup: Flame 0.17.0, libmboard 0.3.1 and OpenMPI 1.6.4.

The experiments explained in following sections were carried out running in independent/separate MPI processes per core.

5.2.2 Activation Mechanism & Agent Migration

In this subsection we present the experimental results of the activation mechanism and agent migration overhead of the proposed performance methodology. The main objective of this set of experiments is to demonstrate the impact of migrating agents. Also, the results show that, despite the inherent high imbalance degree associated to these simulations, it is possible to reduce the degree of imbalance in SPMD ABMS applications.

The following experiments correspond to simulate the SIR model in two scenarios of space dimensions 650x650 and 1000x1000 with initial populations of 30000 and 50000 agents respectively (A and B scenarios) and, in both, cases 10 agents initially infected. The rest of the ABMS environment parameters can be found in [70].

The experiments were performed during 200 simulation steps and, the agents were distributed via Flame round-robin distribution (FlameRR). Thus, regardless of the number of processing elements (PEs) and initial population, the initial number of agents per PE is similar. The Cluster IBM with 128 PEs in 128 cores was used in these experiments with Flame 0.16.2 and libmboard 0.2.1.

For testing purposes, tuning decisions have been performed through a simple LB schema that does not consider communications [70], which is the earliest version of the methodology proposed in this thesis. These experiments utilise the same activation mechanism introduced in this thesis during the monitoring phase. The mechanism is enabled after the fifth simulation step in order to let the workload imbalance get higher before enabling the schema. Additionally, agent migration is launched after the tuning phase but, since the computing measurements immediately vary after the agent migration, the activation mechanism is disabled in the subsequent iteration and enabled again after this. Additionally, the activation mechanism is tested using three imbalance tolerance values: 0.3(30%), 0.15(15%) and 0.05(5%).

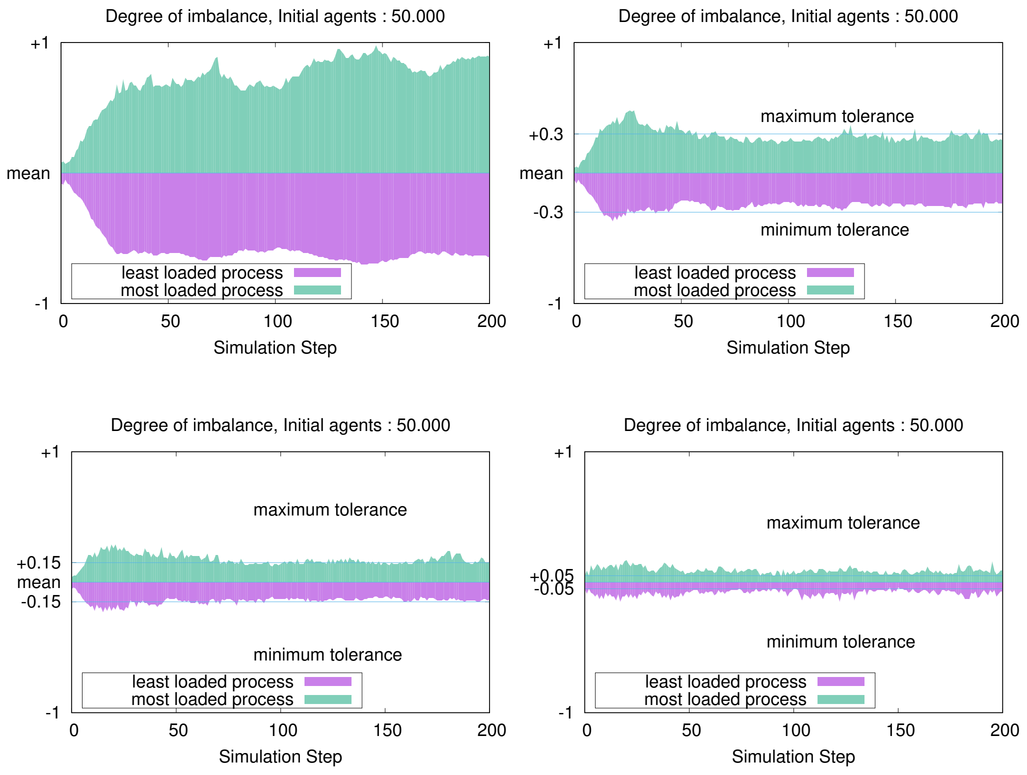

Figure 5.4: Degree of computing imbalance varying the tolerance value.

The first experiment contributes to understand the workload variability of these kinds of applications and the impact of the frequency of launching the simple LB schema. The experiment shows the FlameRR degree of imbalance (without tolerance range) per iteration compared to the degree of imbalance after balancing using different tolerance values. In Figure 5.4, bars closer to 0 (or shorter bars) means lower imbalance degree, the imbalance degree in green colour depicts the most loaded PE in a determined iteration and purple colour depicts the least loaded PE. Through these figures, is shown that the degree of imbalance decreases when the tolerance is increased, but the LB schema is triggered more frequently as a result of the workload variability of these ABMS applications.

The second experiment contributes to understand the agent migration overhead composition and the impact of applying LB schemas in order to improve the performance of this kind of applications. The experiment shows the FlameRR degree of imbalance per iteration compared to the degree of imbalance after balancing using different tolerance values. Table 5.1 summarises the agent migration overhead and the amount of kilobytes transferred (KiB) for different tolerance values. Packing (pack), communication (comm) and unpacking (unpack) correspond to the sum of the maximum times (in seconds) for each iteration. KiB consists of the sum of all agents exchanged during all agent migrations. Consequently, these values expose that the agent migration overhead is greater when reducing the tolerance values because less agent reconfigurations are required. For this reason, better results in terms of execution time are related to find a tolerance value which does not imply an excessive overhead, that to say, a trade-off between the permitted degree of imbalance and the amount of agents migrated.

scenario	30000 agents				50000 agents			
tolerance	pack	comm	unpack	KiB	pack	comm	unpack	KiB
0.05	0.0028	0.280	0.0017	571	0.0064	0.324	0.0047	1252
0.15	0.0035	0.233	0.0021	456	0.0073	0.153	0.0059	689
0.30	0.0039	0.205	0.0027	387	0.0054	0.125	0.0043	544
FlameRR	-	-	-	-	-	-	-	-

Table 5.1: Agent migration overhead.

Tables 5.2 summarises the execution time, the LB overhead and the gain percentage for different values of tolerances. Here, the LB times were calculated by summing the maximum times (in seconds) for each iteration. Concluding from Table 5.1, the agent migration is mainly affected by the amount of bytes transferred (exchanging agents). Here, both scenarios have better results using the LB schema than the FlameRR version. Moreover, in most cases if the imbalance tolerance is reduced the improvement is better but this inversely introduces an aggregate cost for exchanging bytes during the agent migration.

scenario	30000 agents			50000 agents		
tolerance	computing	gain(%)	LB time	computing	gain(%)	LB time
0.05	69.47	40,89	0.29	340.97	36,24	0.35
0.15	74.23	36,84	0.24	369.33	30,94	0.16
0.30	81.96	30,26	0.21	405.16	24,24	0.13
FlameRR	117.53	-	-	534.81	-	-

Table 5.2: Application performance overview including LB schema from [70].

From these experiments, the computational and communicational imbalances that appear during the entire simulation are shown. For both scenarios, the LB schema improves simulation performance reducing the execution time. And, thanks to the constantly workload evaluation of the activation mechanism, the LB schema keeps quite stable the application imbalance degree. Additionally, the overhead is mainly caused by the amount of bytes exchanged during the agent migrations.

5.2.3 Agent System Representation & Message Management

This subsection shows the advantages of utilising our Agent System Representation (ASR) approach and the associated message management in a real SPMD platform. The executions include an improved version of the CTG model presented in [59], which optimises the creation of new agents by decentralising the new agent creation operation resulting in more even workloads and a substantial reduction of the amount of data communicated during this operation.

There is not tuning phase in these experiments, but different initial grid reorganisations in order to re-accommodate the agents according to their spatial location. The reorganisation is performed after the Flame partitioning and before starting the simulations steps, i.e., the AB simulation starts with agent locally grouped. Additionally, we also include grouping agents in accordance with the ASR construction and the hypergraph modelling.

In this case, the Zoltan graph partition library [15] is used for partitioning the hypergraph and deciding a more suitable initial partition. The ASR hypergraph is provided to Zoltan functions for determining an appropriated partition. The grids reallocation, modelled as vertices, is performed according to Zoltan Parallel Hypergraph and Graph partitioner (PHG) [38].

PHG performs the parallel hypergraph partitioning and returns the vertex (grids) ids that should be moved to other PEs in order to balance the load. With the PHG output, some grids will be migrated through migrating all the agents related to them.

PHG can be configured as *Partition*, *Repartition*, or *Refine*. The Partition mode (PhgPA) does not take into account the current vertices distribution (a partition from scratch). The Repartition mode (PhgRP) considers the current vertices distribution for repartitioning the hypergraph. Finally, the Refine mode (PhgRE) refines the given distribution minimising the number of changes.

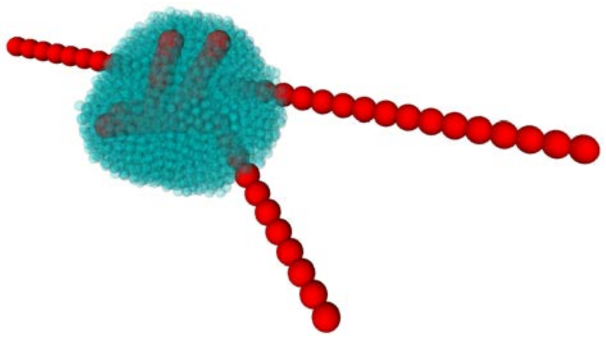

Figure 5.5: Colorectal Tumour Growth model.

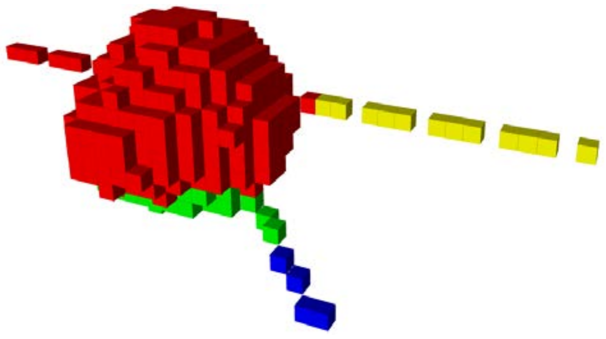

Figure 5.6: Flame geometric grids distribution (4 PEs).

Figure 5.5 shows a graphical representation of the CTG model for 3232 agents wherein green and red spheres represent TC and TEC agents respectively. Figures 5.6, 5.7 and 5.8 show examples of the resulting 3D grids from an initial Flame round-robin (FlameRR), Flame geometric (FlameGeo) and PhgPA partition for four PEs.

The partitioning methods were executed statically from 32 up to 128 processors in Cluster IBM having one process per core. Moreover, the experiments were

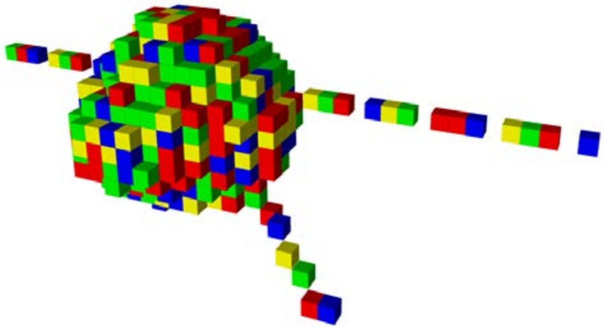
Figure 5.7: Flame round-robin grids distribution (4 PEs).

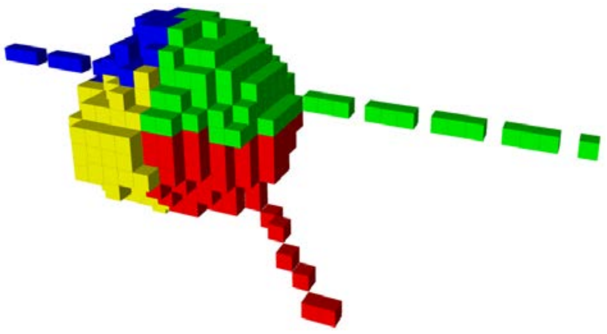
Figure 5.8: PHG grids distribution (4 PEs).

also executed utilising Flame 0.17.0 and libmboard 0.3.1. In these experiments, 15888 agents are simulated within a limitless simulation space for 10 iterations. PHG has been configured with a 3% of imbalance deemed acceptable and partition mode PhgPA. The PHG is executed from the resulting FlameRR and FlameGeo grids.

The ASR grid_size is defined as 50 units (microns) which corresponds to the double of the reach value of the cells messages (25 microns) and also results in a manageable amount of vertices and hyperedges when the hypergraph is generated.

Table 5.2.3 shows the percentage of messages that are held in the sender PE and those that are dispatched to external PEs. It shows that the amount of messages retained in the sender PE impacts the performance when the agents are organised according to their interaction regions. Due to the FlameRR experiments do not organise the agents according to their interaction, this agent distribution shows the worst results. The FlameGeo experiments have more percentage of messages held than the PHG cases because the geometric method does not

	FlameGeo		FlameRR		PHG-FlameGeo		PHG-FlameRR	
	intern	extern	intern	extern	intern	extern	intern	extern
32 p.	92.89	7.11	7.77	92.23	78.75	21.25	78.58	21.42
64 p.	94.47	5.53	14.09	85.91	82.71	17.29	82.74	17.26
128 p.	95.74	4.26	24.57	75.43	85.89	14.11	85.92	14.08

Table 5.3: Percentage of messages held (intern) and messages dispatched (extern).

generate even partitions in terms of grids. Moreover, this method produces many empty PEs, and hence the grid receptors have more possibilities to be into the sender PE. Finally, we note that a small amount of messages dispatched directly impacts the performance of the neighbouring PEs because the external PEs have to examine a greater amount of information and, later on, determine its significance.

5.2.4 Overall Tuning Improvements

This subsection focuses on simulating a real AB model, in this case CTG and KCF models, and also analysing the methodology performance improvements, denoted as Dynamic Load Balancing (DLB) method in figures, in a real SPMD platform.

The first experiment includes all the Flame enhancements working together as well as the improved version of the CTG model used in the previous subsection. The ASR grid_size is also defined as 50 units (microns).

The tuning decisions rely on Zoltan [15] which is, to the best of our knowledge, the only well-established partition library supporting the possibility of being used at runtime. Zoltan parts the hypergraph utilising the Parallel Hypergraph and Graph partitioner (PHG), this was also explained in previous subsection. PHG returns the vertex (grid) ids that need to be migrated.

In order to reduce the time to find an appropriate graph partition, the PHG accuracy has been set up to 0.2 (20%) of imbalance deemed acceptable. The experiments performed 20 simulation steps and the agents were initially distributed with PhgPA mode, except for Flame round-robin (FlameRR) and geometric (FlameGeo) distributions.

The experiments correspond to simulate the CTG model within a limitless simulation space and an initial population of 63505 agents, 49 initial TECs. The Cluster IBM with 128 PEs in 128 cores was used in these experiments with the Flame 0.17.0 and libmboard 0.3.1. Additionally, the activation mechanism is configured with 0.3 tolerance value (30% imbalance acceptable) and enabled after the first simulation step. Since the workload immediately varies after performing an agent migration, the activation mechanism is kept disabled until the iteration subsequent to the tuning succeeding iteration.

Approach	Tuning calls	Vertices(avg)	Overhead(sec)	Exec time(sec)
FlameGeo	-	-	-	26218.9
FlameRR	-	-	-	1038.1
Static PhgPA	-	-	-	881.3
DLB PhgRP	10	3746	83.3	741.3
DLB PhgPA	10	3743	63.4	710.7
DLB PhgRE	9	3745	56.9	699.7

Table 5.4: Methodology execution time overview.

Table 5.4 shows the execution times of our methodology by comparing different PHG options with three static approaches (FlameRR, FlameGeo and initial PhgPA). Our approach obtains much better results than the static approaches. FlameGeo leads to the highest time because it divides the space into orthogonal rectangles, creating uneven or empty partitions according to the agents' spatial locations, while FlameRR randomly distributes the agents generating a similar number of agents per PE. The PHG versions gain more than 30% over the FlameRR case in terms of execution time, even an initial Static PhgPA partitioning improves the Flame times. In addition, the hypergraph options show similar results and the main difference relies on the overhead time. For all versions, the number of tuning calls and average number of vertices are similar. Nevertheless, repartitioning the current partition (PhgRP) is expensive compared to partitioning from scratch (PhgPA), and refining the hypergraph (PhgRE) is the best approach for these experiments.

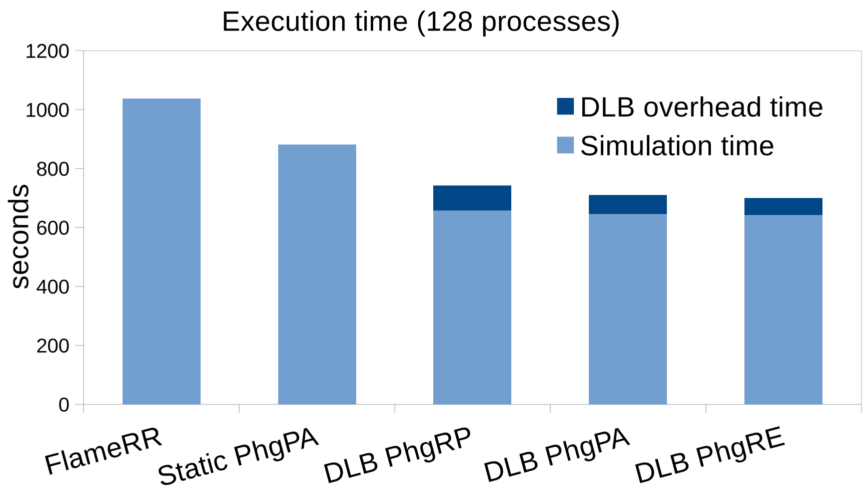

Figure 5.9: Execution times of different PHG options and static approaches.

Figure 5.9 depicts the execution time and overhead shown in Table 5.4. For all versions, our approach significantly reduces the execution time. FlameGeo has been excluded because of its excessive execution time.

	% Message Management		Grids Construction	PHG Partitioning	Agent Migration	Total Time
	intern	extern				
PhgRP	95.95	4.05	10.3	70.1	2.9	83.3
PhgPA	96.57	3.43	7.3	54.0	2.1	63.4
PhgRE	96.53	3.47	8.8	45.1	3.0	56.9

Table 5.5: Methodology overhead overview.

Table 5.5 shows the composition of the methodology overhead and the message management percentages. The main affectation is rooted in the time required by PHG for partitioning the hypergraph. Furthermore, repartitioning the current partition (PhgRP) is more expensive compared than partitioning from scratch (PhgPA), and refining the hypergraph (PhgRE) is the best approach for these experiments. Even so, the percentage of messages sent to external PEs are kept low, that is to say, through our ASR the major part of the agent messages is retained in source PEs (sender). In fact, dispatching a small amount of messages impacts directly the performance of the recipient PEs because these have to examine a smaller amount of messages and, later on, determine its significance.

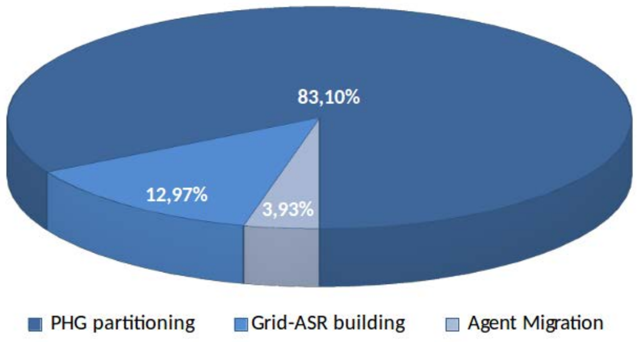

Figure 5.10: Load balancing average overhead.

Figure 5.10 shows that most of the overhead out the methodology comes from partitioning the hypergraph using PHG. These results suggest that reducing the number of vertices by increasing the *grid_size* could reduce the PHG overhead.

5.2.5 Methodology Overhead Enhancements

In the second experiment, the KCF model is executed including our Flame enhancements too. In this cases 3000 agents are simulated for 500 simulation steps in Marenostrum III using 128 PEs in 8 computing nodes. The experiment demonstrates that increasing the grid_size the PHG overhead is reduced, and consequently, our methodology intrusion.

The experiment corresponds to simulate the KCF model within a limitless simulation space and two grid_size values, 10 (messages reach) and 100 units (nanometres). The PHG accuracy has been set up to 0.25 (25%) of acceptable imbalance and the partitioning mode to Refine (PhgRE).

The activation mechanism is configured with 0.15 tolerance value (15% imbalance acceptable) and enabled after the first simulation step. The activation mechanism is kept disabled until the iteration subsequent to the tuning succeeding iteration.

Table 5.6 shows the execution times of our methodology using PHG in Refine mode (PhgRE) with two grid_size values 10 (DLB PhgRE) and 100 (DLB PhgRE+) nanometres. Also, the original Flame static partitioning methods are compared (FlameRR and FlameGeo). As was proven in previous subsection, the methodology presented in this thesis obtains much better results than the static approaches. Flame approaches obtain high execution times because, in the FlameGeo case, generates uneven or empty partitions and, in FlameRR case, cyclically distributes the agents without any locality consideration.

Approach	Tuning calls	Vertices		Overhead(sec)		Exec time(sec)
		max	avg	max	total	
FlameGeo	-	-	-	-	-	9529
FlameRR	-	-	-	-	-	4795
DLB PhgRE	166	61931	12290	18.60	500.95	1916
DLB PhgRE+	250	2951	546	8.88	387.40	1696

Table 5.6: Methodology execution time detail.

In this case, our approach significantly reduces the execution time. The DLB PhgRE and DLB PhgRE+ versions gain 60% and 65% (respectively) with respect of the static FlameRR time. The overhead times of both PHG cases are similar but the DLB PhgRE+ version, with larger grid_size, presents fewer number of tuning calls, maximum and average number of vertices. Having a larger grid_size reduces the hypergraph partitioning associated overhead, but unfortunately the ASR and the hypergraph modelling lose quality which results in an increment of the frequency of tuning phase calls. Therefore, a larger grid_size also generates loosing quality of the agent system representation but the hypergraph partitioning overhead is reduced. On the contrary, reducing the grid_size the hypergraph grows because the ASR generates more grids with more interconnections. This provides a finer analysis of the workload but results in a heavier partitioning operation. In spite of this, the methodology always substantial improves the performance of this application.

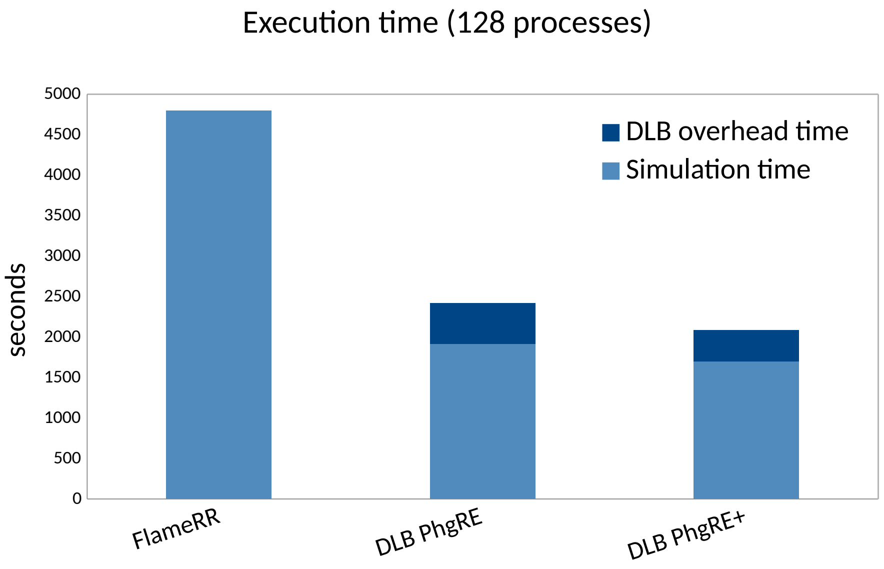

Figure 5.11: Execution times for two ASR grid_size.

Figure 5.11 shows the execution time and overhead from Table 5.6. FlameGeo has been excluded from this figure because of its excessive execution time.

For this reason, better results in terms of execution time are related to find a tolerance value which does not imply an excessive overhead, that to say, a trade-off between the permitted degree of imbalance and the amount of agents migrated.

There are several variations of graph partitioning, naturally, there is a trade-off between runtime and solution quality [88] and, in such cases, the runtime of the hypergraph partitioning algorithms is much higher than the one of graph partitioning [84].

In conclusion, there is a trade-off between PHG runtime and solution quality. This indicates that depending on the expected quality solution, finding a proper grid reconfiguration may take longer than expected. This trade-off between runtime and solution quality is also an intrinsic feature of graph and hypergraph partitioning algorithms which are usually affected by, among other things, the problem size, hypergraph complexity, number of PEs and the partitioning algorithm efficiency.

5.3 Summary

In this chapter, we have presented the evaluation of the hypergraph-based methodology to reconfigure the workload of SPMD ABMS applications through migrating agents. The evaluation is carried out through a set of experiments using real AB models in a real platform. The experiments include all the implementation enhancements described in the previous chapter.

This chapter also demonstrates that this kind of applications require dynamic load balancing mechanisms to improve their performance and make more efficient utilisation of the HPC resources. Consequently, we obtain significant performance gains using our approach.

To do so, three real AB models have been considered: *Susceptible-Infected-Remove*, *Colorectal Tumour Growth* and *Keratinocyte Colony Formation.* Such models develop dynamic changes in terms of workloads grounded in random rates of creation and elimination of agents. Additionally, the environments wherein the experiments were carried out (*Cluster IBM* and *Marenostrum III*) are summarised.

Moreover, the agent migration is analysed and later the advantages of using our ASR approach to improve the message management mechanism. Subsequently, the overall methodology is analysed, showing that the associated overhead is rooted in trade-off between hypergraph partitioning runtime and solution quality.

Finally, evaluating different aspects of the methodology presented in this thesis, as well as an integral whole, the significant performance gains and advantages utilising our approach are demonstrated.

The last chapter of this thesis describes the conclusions obtained throughout this research.

Chapter 6

Conclusions

In this thesis, we have introduced the Agent based Modelling and Simulation (ABMS) applications in High Performance Computing (HPC) environments. Within this scope, ABMS applications usually show a dynamic workload variability that negatively affects the performance of the application and experiencing imbalanced computing/communications among the processing elements (PE).

In this sense, we presented the motivation, design, implementation and evaluation of the contribution of this work: methodology that enables dynamic performance enhancements for HPC ABMS applications. The goal of the methodology is to reduce the total execution time of HPC ABMS applications, as well as augmenting the efficient use of computational resources. As a result, the application will be able to process a large number of agents with complex rules as fast and efficiently as possible.

In the design of the methodology, we have considered:

- ABMS applications with spatially-explicit models with neighbouring communications, which are commonly utilised to model real-world spatial data for studying complex spatial systems. In these models, agents are associated with a spatial location and communication dependencies can be determined according to a spatial distance function.
- ABMS applications designed under the Single Program Multiple Data (SPMD) structure which is the dominant programming model for large-scale applications and the most common application structure for HPC ABMS platforms.
- Existing parallel hypergraph partitioning tools to efficiently assist tuning decisions in order to decide a proper workload distribution. Such tools, provide hypergraph partitioning methods that enable partitioning complex and large problems which usually cannot fit in the memory, cost too much to partition and are also difficult to implement.
- PEs allocate separate MPI processes in such a way that independents PEs

execute a unique MPI process during the entire simulation.

The methodology introduces a strategy to reduce the gaps of the computing and communication workloads between PEs as the simulation proceeds. The methodology adjusts the global simulation workload migrating groups of agents among the PEs according to their computation workload and their message connectivity map modelled using a hypergraph. The hypergraph is lastly partitioned to decide a proper workload distribution. Consequently, the methodology tunes dynamically the execution of ABMS applications in order to minimise the computing and communication workloads imbalances.

In this way, it is possible to execute faster agent-based (AB) simulations of a large number of agents with complex interaction rules. Other contributions included in this work that we consider significant to mention are:

- A description of the components required to define a dynamic load balancing (DLB) approach in ABMS applications. The main components described are: agent system representation (ASR) to model the agents' workload and interactions, tuning decisions to determine proper performance adjustments and agent migration to transfer agents between PEs. These components, along with workload monitoring and measurement, contribute to control and minimise the unexpected workload imbalances introduced by the agents.
- A monitoring schema that measures the parallel application workload at runtime to identify performance problems in the PEs and an activation mechanism that evaluates the imbalance impact according to a tolerance value and decides, when necessary, applying the load balancing strategy.
- An ASR through a grid-based spatial organisation that characterises the agent locations and workloads in relation to their PE which also helps to minimise the CPU/memory resource requirements and allows modelling indefinitely large domains.
- A weighted hypergraph based point of view of the agent workload (vertices) and interactions (hyperedges) which provides the analysis and identification of imbalances spots within the groups of agents composition (grids). In this manner a hypergraph partitioning algorithm is utilised to calculate a more balanced grid distribution.
- An agent migration schema that mitigates its runtime impact through migrating groups of agents, minimising the number of transferences and reducing the communication latency by communication/computing overlapping.
- A suitable message management definition based on the grid-based spatial organisation that enables determining the internal PE messages and the messages that need to be dispatched to other PE.
- A detailed implementation of our DLB methodology in a real HPC ABMS platform (Flame) and tested with three real AB models Susceptible-Infected-

Remove (SIR), Colorectal Tumour Growth (CTG) and Keratinocyte Colony Formation (KCF).

Next, the chapter presents conclusions derived from this thesis. We also describe the open lines that can be considered in the future in order to provide further performance strategies and improvements in the area of dynamic tuning of SPMD ABMS applications.

6.1 Final Conclusions

Nowadays, thanks to the accessibility to computational resources, the HPC environments solve computing problems when these become more complex and the amount of required computing power increases. In such problems, decomposing the problem into parallel program may be the only way to achieve solutions in reasonable time. There is a group of problems wherein the HPC applications dynamically change their behaviour during the execution and hence its performance index changes too.

Here, the initial decomposition would likely remain as an efficient data distribution for a limited time. In addition, these applications need to be dynamically assisted by specialised strategies at runtime in order to avoid an excessive computing/communication workloads imbalance as the simulation proceeds. HPC ABMS applications belong to this category because they usually show a dynamic workload variability that negatively affects the performance of the application and thus increases the runtime. Therefore, the HPC ABMS applications also need a dynamic load balancing (DLB) strategy, so that emerging imbalanced problems can be solved during execution.

In this regard, the overall contribution of this thesis enables reconfiguring the computing and communication workload of SPMD ABMS applications for spatially-explicit models with neighbouring agent communications.

Our approach is based on building three main components: the agent system representation (ASR), the tuning decisions and the agent migration. The ASR has to provide useful information to diagnose the workload imbalance spots and help to identify the relation between such computing and communication volumes with the agents and the grid structure. It has also to provide information relating to agent/grid spatial locations and their containing PE with respect to rest PE resources. The tuning decision correspond to appropriate tuning rules based on hypergraph partitioning, to deal with the imbalance problem. The agent migration is the mean whereby the ABMS platform is capable of transferring agents/grids from a PE to another in order to modify the workload of such PEs and consequently, affecting the global workload status of the simulation.

In simulations with large number of agents with complex interaction rules and agent creation/elimination behaviours, in term of resources, the CPU/memory requirements for storing such information may also become unmanageable. Additionally, in some cases, the agent simulation performs massive creation of new agents leading to an indefinitely domain expansion, otherwise, the elimination of agents shrinks the domain.

In the same manner, we propose creating grids only for the space occupied by agents in order to reduce these excessive CPU/memory requirements, our approach idea is taken from the known technique named rasterisation. The defined clustering algorithm is rooted on this principle, so the grids are only created according to the space occupied by agents.

Consequently, we propose an ASR based on clustering the spatial regions during the simulation into grids. As a result of clustering the ABMS domain with our grid method, sets of agents are associated to a unique virtual grid and their interactions and workload measurements are also analysed in terms of groups of agents (grids). In this manner, it is possible to reduce the computational complexity and the CPU/memory requirements.

Additionally, we introduce that the joint measurements and exchanges between agents contained in a grid can be modelled as a weighted hypergraph representation which allows more accurately modelling the agents system interactions. Hereinafter, we propose making tuning decisions grounded in partitioning the weighted hypergraph into equally balanced partitions through integrating a hypergraph partitioning tool. Thereafter, the agent migration is undertaken in order to fulfil the grids reconfiguration decisions (the hypergraph resulting new partitions) and adjust the global workload. Unfortunately, the runtime of hypergraph partitioning algorithms is intrinsically conditioned by, among other things, the problem size, hypergraph complexity, number of PEs and the partitioning algorithm efficiency. As was mentioned along with this thesis, there is a trade-off between the hypergraph partitioning runtime and solution quality. This indicates that, depending on the expected quality solution, finding a proper grid reconfiguration may take longer than expected.

In the same way, the amount of agents migrated has also an associated overhead mainly caused by the amount of bytes exchanged during the agent migrations. In this regard, there is also a trade-off between the frequency of tuning phase calls and the amount of agents migrated.

The message management mechanism defined in this thesis operates directly over the ASR definition and assists the platform to determine the recipient PEs related to the agents' outgoing messages across all PEs. In this sense, the percentage of messages sent to external PEs (receivers) are kept low, that is to say, through our ASR the major part of the agent messages is retained in source PEs (senders). We note that the amount of messages dispatched directly impacts the performance of the neighbouring PEs because receiver PEs have to examine a smaller amount of received messages.

Finally, we evaluate our hypergraph-based methodology through a set of experiments using real AB models in a real HPC platform. In conclusion, evaluating different aspects of the methodology presented in this thesis, as well as an integral whole, the significant performance gains and advantages utilising our approach have been demonstrated. For the Colorectal Tumour Growth and Keratinocyte Colony Formation models, our tuning methodology gains more than 30% and 60% (respectively) in terms of execution time with respect of the round-robin distribution (the best Flame approach in the experiments).

6.2 Future Work and Open Lines

The work presented in this thesis gives rise to a wide range of affordable open lines and further work. We have classified some relevant topics that need to be considered in order to extend the scope of this research:

- Determine the best number of PEs. The number of PEs being used is previously decided, but this can be adjusted to reduce possible inefficiencies (PEs with longer periods of idleness). In an ideal scenario, where there is no communication overhead or time constraints given by agent operation/states, adding more resources will scale well. However, this situation is not common for these applications. These performance issues can be mitigated by developing a solution for adapting the number of PEs used to provide an efficient execution in order to obtain the lowest execution time possible using the available resources.
- Define a strategy to dynamically determine the best grid size. The choice of a proper ASR grid size significantly affects the performance of CPU/memory requirements, hypergraph partitioner overhead, message management performance and the ASR workload modelling quality. For this reason, finding the best grid (hence, expected solution quality) in each specific workload case is a difficult task and very relevant to significantly reduce the overall overhead.
- Tune the tolerance range. Within the monitoring phase, the tolerance range determines the permitted imbalance bounds in terms of iteration time and consequently indicates the frequency of launching the tuning phase. As in the previous case, finding the best tolerance value for each specific case, for a specific ABMS problem with a particular workload variability, will reduce the application runtime and minimise the DLB overhead.
- Extend to master-worker paradigm. There are several HPC applications which are designed utilising the master-worker program structure, where several load balancing studies take place. The methodology presented in this thesis can be extended to this paradigm by placing the DLB approach in the master PE which should be responsible for collecting the global ASR

and giving instructions for migrating agents.

- Include ASR support for non spatially-explicit AB models. It is noteworthy that other options to implement ABMS exist. There are agent definitions without spatial representation at all, but the agents are linked together into a network in which the only indication of an agents relationship to other agents is the list of the agents to which it is connected by network links. In such cases, the ASR has to be suited for applying a different clustering algorithm and agents' relations interpreting procedure. For the clustering algorithm, a different relation function or sort criterion has to be defined in order to understand the agents connection characteristics for grouping agents.
- Test with other AB models and platforms. There is a wide variety of AB models and other ABMS platforms intended to HPC environment to extend the test of our approach. Since AB models defined with different behavioural patterns and other ABMS platforms manage agent computing and interaction with different functioning concepts, these extended tests will provide significant key aspect to a wider range of models and porting our implementation to other platforms.
- Explore other graph/hypergraph partitioning methods. Following the previous idea, there are varied graphs/hypergraph partitioning algorithms usually utilised for solving domain dependent optimisation problems modelled in terms of weighted or unweighted hypergraph/graphs. We consider relevant to explore other approaches in order to analyse and propose a solution for reducing the partitioning overhead associated.
- Perform scalability tests. Since applications express well balanced data decomposition and full parallelism on their codes, the performance is not always linear to the number of resources, as expected. For evaluating this aspect, it is necessary to study the different factors limiting the application scalability, either testing our methodology with a wider number of PEs or redefining certain components and the implementation of our DLB.
- Include large scale simulations. The large amount of biologic, statistics, scientific and engineering problems - just to mention a few - that can be studied and solved thanks to the computational resources existing today, led to the design of more complex AB models and computing power. In this sense, another research line is evaluating, adapting and applying our methodology on systems architectures with high scalability, which can be designed with a different memory hierarchy and a huge amount of PEs.

6.3 Ph.D. Internship

During the course of this thesis a Ph.D. internship was done at Royal College of Surgeons in Ireland (RCSI). The objective was analysing the performance of a Colorectal tumour model and exploring large simulation capabilities. The internship leads to a journal publication associated to the European project ANGIOPREDICT (Predictive Genomic Biomarkers Methods for Combination Bevacizumab 'Avastin' Therapy in Metastatic Colorectal Cancer, http://www.angiopredict.com/).

6.4 List of Publications

The work presented in this thesis has reported the following publications:

- **C. Márquez, E. César, J. Sorribes, "Generación Automática de Funciones de Migración de Agentes en FLAME", XXIII Jornadas de paralelismo (JP), Elche, Spain, September 2012.**

  This work focuses on the automatic generation of migration routines in Flame using the SIR model. We showed that it is possible to make workload modifications through migrating agents between PEs.

- **C. Márquez, E. César, J. Sorribes, "A Load Balancing Schema for Agent-based SPMD Applications", Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, USA, July 2013.**

  This work proposes a schema that dynamically adjusts the workload and migrates agents that only consider computing workload among PEs. In this case, a minimalist ASR is built considering the joint agent workload and the quantity of agents for each PE.

- **C. Márquez, E. César, J. Sorribes, "Agent Migration in HPC Systems using FLAME", 1st Workshop on Parallel and Distributed Agent-Based Simulation (PADABS), satellite Workshop of Euro-Par, Aachen, Germany, August 2013.**

  In this paper, we published an efficient migration mechanism for migrating agent between PEs and analyse the performance factors that affect the overhead when migrating agents. We also presented a Flame template to generate migration routines.

- **C. Márquez, E. César, "Tutorial: Agent-based Simulations using Flame", Social Simulation Conference (SSC), Barcelona, Spain, September 2014.**

This is not a publication in itself, but a hands-on tutorial given in SSC'14 for AB modellers concerned with scalability issues, whose requirements could be fulfilled by the power of HPC.

- **C. Márquez, E. César, J. Sorribes, "Impact of Message Filtering on HPC Agent-Based Simulations", pp. 65-72, Proceedings of 28th European Simulation and Modelling Conference ESM'2014, 28th European Simulation and Modelling Conference (ESM), Porto, Portugal, October 2014.**

  In this work, we published a message filtering mechanism based on the ASR presented in this thesis and we present a Flame template for automatically generating message filtering routines. This paper shows the performance gains of such message mechanism with the CTG model. This work is aiming to overcome the reduction of the excessive agents' messages applied to AB simulations.

- **C. Márquez, E. César, J. Sorribes, "Graph-Based Automatic Dynamic Load Balancing for HPC Agent-Based Simulations", 3rd Workshop on Parallel and Distributed Agent-Based Simulation (PADABS), satellite Workshop of Euro-Par, Vienna, Austria, August 2015.**

  This paper introduces a DLB strategy that tunes the global simulation workload migrating groups of agents among the PEs according to their computation workload and their message connectivity map modelled using a hypergraph. This structure is build based on clustering agents into grids. This Hypergraph is partitioned using the Zoltan Parallel Hypergraph partitioner method (PHG).

- **Guiyeom Kang, Claudio Márquez, Ana Barat, Annette T. Byrn, Jochen H.M. Prehn, Joan Sorribes, Eduardo César. "Colorectal tumour simulation using agent based modelling and high performance computing", Future Generation Computer System (FGCS), April 2016.**

  This journal publication shows the design and implementation of the DLB mechanism including required tuning features, such as agent migration, message filtering, agent connectivity mapping and performance measurement monitoring, in order to improve the performance of parallel AB simulations. The mechanism is implemented in Flame with the CTG simulation model presented in this thesis.

6.5 Special Acknowledgements

This work has been partially supported by MICINN-Spain under contract TIN2011-28689-C02-01 and TIN2014-53234-C2-1-R and GenCat-DIUiE(GRR) 2014-SGR-576.

This research was also funded by the European Community's Framework Programme Seven (FP7) Programme under contract No. 278981 AngioPredict and supported by the DJEI/DES/SFI/HEA Irish Centre for High-End Computing (ICHEC) and the Barcelona Supercomputing Center-Centro Nacional de Supercomputación (BSC-CNS).

In addition, the authors thankfully acknowledge the resources and technical assistance provided by:

- Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, Barcelona, Spain.
- Centre for Systems Medicine and Physiology & Medical Physics Department, Royal College of Surgeons in Ireland (RCSI), Dublin 2, Ireland.
- Computer Applications in Science & Engineering Department, Barcelona Supercomputing Center-Centro Nacional de Supercomputación (BSC-CNS), Barcelona, Spain.

Bibliography

[1] Top500 supercomputer site [online]. http://www.top500.org.

[2] Shahnaz Afroz, Hee Yong Youn, and Dongman Lee. Performance of message logging protocols for nows with mpi. In *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, PRDC '99, pages 252–, Washington, DC, USA, 1999. IEEE Computer Society.

[3] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[4] César Allande, Josep Jorba, Anna Sikora, and Eduardo César. A performance model for openmp memory bound applications in multisocket systems. *Procedia Computer Science*, 29:2208 – 2218, 2014.

[5] George S. Almasi and Allan Gottlieb. *Highly parallel computing (2. ed.).* Addison-Wesley, 1994.

[6] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.

[7] Angiopredict. Predictive genomic biomarkers methods for combination bevacizumab (avastin) therapy in metastatic colorectal cancer ANGIOPREDICT,. EU's Framework Programme Seven (FP7) under contract 306021, 2014.

[8] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice Hall Professional Technical Reference, 1982.

[9] C. Berge. *Hypergraphs: Combinatorics of Finite Sets.* North-Holland, 1989.

[10] Pavel Berkhin. Survey of clustering data mining techniques. Technical report, 2002.

[11] M. Bithell and W.D. Macmillan. Escape from the cell: Spatially explicit modelling with and without grids. *Ecological Modelling*, 200(12):59 – 78, 2007.

[12] Vicente Blanco. *Análisis, predicción y visualización del rendimiento de métodos iterativos en HPF y MPI.* PhD thesis, Universidad de Santiago de Compostela, 2002.

[13] Douglas M. Blough and Peng Liu. Fimd-mpi: A tool for injecting faults into mpi applications. *Parallel and Distributed Processing Symposium, International*, 0:241, 2000.

[14] Cristina Boeres, Vinod E. F. Rebello, Cristina Boeres, and Vinod E. F. Rebello. Towards optimal static task scheduling for realistic machine models: theory and practice, 2003.

[15] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[16] Jurgen Brehm, Patrick H. Worley, and Manish Madhukar. Performance modeling for spmd message-passing programs, 1996.

[17] Alain Bretto. Introduction to hypergraph theory and its use in engineering and image processing. volume 131 of *Advances in Imaging and Electron Physics*, pages 1 – 64. Elsevier, 2004.

[18] Alain Bretto. *Hypergraph Theory: An Introduction.* Springer Publishing Company, Incorporated, 2013.

[19] Alain Bretto, Yannick Silvestre, and Thierry Vallée. Cartesian product of hypergraphs: properties and algorithms. In *Proceedings Fourth Athens Colloquium on Algorithms and Complexity, ACAC 2009, Athens, Greece, August 20-21, 2009.*, pages 22–28, 2009.

[20] Daniel G. Brown, Rick Riolo, Derek T. Robinson, Michael North, and William Rand. Spatial process and data models: Toward integration of agent-based models and gis. *Journal of Geographical Systems*, 7(1):25–47, 2005.

[21] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[22] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing Principles and Paradigms.* Wiley Publishing, 2011.

[23] E. Cesar, A. Moreno, J. Sorribes, and E. Luque. Modeling master/worker applications for automatic performance tuning. *Parallel Comput.*, 32:568–589, September 2006.

[24] Eduardo Cesar. *Definition of Framework-based Performance Models for Dynamic Performance Tuning.* PhD thesis, Universitat Autonoma de Barcelona. Departament d'Arquitectura de Computadors i Sistemes Operatius, 2006.

[25] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? *SIGOPS Oper. Syst. Rev.*, 28:61–73, November 1994.

[26] Xu Chengzhong and Francis C. M. Lau. A survey of nearest-neighbor load balancing algorithms. In *Load Balancing in Parallel Computers*, volume 381 of *The Kluwer International Series in Engineering and Computer Science*, pages 21–35. Springer US, 1997.

[27] C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331, July 2008.

[28] S. Coakley, M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough. Exploitation of high performance computing in the flame agent-based simulation framework. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 538–545, June 2012.

[29] Simon Coakley, Paul Richmond, Marian Gheorghe, Shawn Chin, David Worth, Mike Holcombe, and Chris Greenough. *Large-Scale Simulations with FLAME*, pages 123–142. Springer International Publishing, Cham, 2016.

[30] Simon Coakley, Rod Smallwood, and Mike Holcombe. Using X-machines as a formal basis for describing agents in agent-based modelling. In *Proceedings of 2006 Spring Simulation Multiconference*, pages 33–40, April 2006.

[31] Nicholson Collier and Michael North. Parallel agent-based simulation with repast for high performance computing. *Simulation*, 89(10):1215–1235, 2013.

[32] Gennaro Cordasco, Rosario De Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. Bringing together efficiency and effectiveness in distributed simulations: the experience with d-mason. *Simulation*, 89(10):1236–1253, 2013.

[33] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[34] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, and Vittorio Scarano. Distributed load balancing for parallel agent-based simulations. In *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '11, pages 62–69, Washington, DC, USA, 2011. IEEE Computer Society.

[35] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[36] Toh Da-Jun, Francis Tang, Travis Lee, Deepak Sarda, Arun Krishnan, and Andrew Goryachev. Parallel computing platform for the agent-based modeling of multicellular biological systems. In *Parallel and Distributed Computing: Applications and Technologies*, pages 5–8. Springer, 2005.

[37] Chris Greenough David Worth, Shawn Chin. FLAME tutorial examples : a simple SIR infection model. Technical Report RAL-TR-2012-017, Rutherford Appleton Laboratory, November 2012.

[38] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[39] Hank Dietz. Linux parallel processing howto, 1998.

[40] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23:5–16, February 1990.

[41] A. Fahad, N. Alshatri, Z. Tari, A. Alamri, I. Khalil, A. Y. Zomaya, S. Foufou, and A. Bouras. A survey of clustering algorithms for big data: Taxonomy and empirical analysis. *IEEE Transactions on Emerging Topics in Computing*, 2(3):267–279, Sept 2014.

[42] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[43] Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(2):298–305, 1973.

[44] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Faerber. Indexing highly dynamic hierarchical data. *Proc. VLDB Endow.*, 8(10):986–997, June 2015.

[45] P. Fjallstrom. Algorithms for graph partitioning: A survey, 1998.

[46] Steven Fortune. Voronoi diagrams and delaunay triangulations. *Discrete & Computational Geometry*, 1995.

[47] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[48] Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[49] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[50] G. Nigel Gilbert. *Agent-based models.* Quantitative applications in the social sciences. Sage, Los Angeles, CA, 2008.

[51] Nigel Gilbert, Andreas Pyka, and Petra Ahrweiler. Innovation networks - A simulation approach. *J. Artificial Societies and Social Simulation*, 4(3), 2001.

[52] Raymond Greenlaw and Sanpawat Kantabutra. Survey of clustering: Algorithms and applications. *Int. J. Inf. Retr. Res.*, 3(2):1–29, April 2013.

[53] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22:789–828, September 1996.

[54] Per Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency - Practice and Experience*, 5(5):407–423, 1993.

[55] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986.

[56] Jay P. Hoeflinger. Extending openmp to clusters, 2006.

[57] Josep Jorba. *Análisis automático de prestaciones de aplicaciones paralelas basadas en paso de mensajes.* PhD thesis, Universitat Autònoma de Barcelona. Departament d'Arquitectura de Computadors i Sistemes Operatius, 2006.

[58] Amir Kamil and Katherine Yelick. *Hierarchical Computation in the SPMD Programming Model*, pages 3–19. Springer International Publishing, Cham, 2014.

[59] Guiyeom Kang, Claudio Márquez, Ana Barat, Annette T. Byrne, Jochen H.M. Prehn, Joan Sorribes, and Eduardo César. Colorectal tumour simulation using agent based modelling and high performance computing. *Future Generation Computer Systems*, 67:397 – 408, 2017.

[60] George Karypis and V. Kumar. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical Report 97-060, Department of Computer Science, University of Minnesota, 1997. http://www.cs.umn.edu/ metis.

[61] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[62] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011.

[63] B. W. Kernighan and Shunjiang Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal, The*, 49(2):291–307, February 1970.

[64] Tanya Kostova, Tina Carlsen, and Jim Kercher. Individual-based spatially-explicit model of an herbivore and its resource: the effect of habitat reduction and fragmentation. *Comptes Rendus Biologies*, 327(3):261–276, March 2004.

[65] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[66] Anany V. Levitin. *Introduction to the Design and Analysis of Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[67] A. J. Lotka. *Elements of Physical Biology.* Williams and Wilkins, Baltimore, 1920.

[68] Huiwei Lu, S. Seo, and Pavan Balaji. Mpi+ult: Overlapping communication and computation with user-level threads. In *HPCC'15*, New York, 08/2015 2015. IEEE, IEEE.

[69] Charles M. Macal and Michael J. North. Tutorial on agent-based modeling and simulation. In *Proceedings of the 37th Conference on Winter Simulation*, WSC '05, pages 2–15. Winter Simulation Conference, 2005.

[70] Claudio Márquez, Eduardo César, and Joan Sorribes. A load balancing schema for agent-based spmd applications. In *International Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA*, pages 62–69, 2013.

[71] Claudio Márquez, Eduardo César, and Joan Sorribes. Agent migration in hpc systems using flame. In *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 523–532. Springer Berlin Heidelberg, 2014.

[72] Claudio Márquez, Eduardo César, and Joan Sorribes. Impact of message filtering on hpc agent based simulations. In *European Simulation and Modelling Conference 2014*, pages 62–72, 2014.

[73] Claudio Márquez, Eduardo César, and Joan Sorribes. *Graph-Based Automatic Dynamic Load Balancing for HPC Agent-Based Simulations*, pages 405–416. Springer International Publishing, Cham, 2015.

[74] Oliver A. McBryan. An overview of message passing environments. *Parallel Computing*, 20:417–444, 1994.

[75] Dan I. Moldovan. *Parallel Processing: From Applications to Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[76] A. Moreno, E. Cesar, A. Guevara, J. Sorribes, and T. Margalef. Load balancing in homogeneous pipeline based applications. *Parallel Comput.*, 38(3):125–139, March 2012.

[77] nCUBE company. ncube 2 supercomputers parallel programming principles. Technical report, Foster City, CA., 1993.

[78] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1):3, 2013.

[79] A. Osman and H. Ammar. A scalable dynamic load-balancing algorithm for spmd applications on a non-dedicated heterogeneous network of workstations (hnow), 2003.

[80] David A. Papa and Igor L. Markov. Hypergraph partitioning and clustering. In *In Approximation Algorithms and Metaheuristics*, 2007.

[81] Roberto Uribe Paredes, Claudio Márquez, and Roberto Solar. Construction strategies on metric structures for similarity search. *CLEI Electron. J.*, 12(3), 2009.

[82] HazelR. Parry and Mike Bithell. Large scale agent-based modelling: A review and guidelines for model scaling. In *Agent-Based Models of Geographical Systems*, pages 271–308. Springer Netherlands, 2012.

[83] H. Van Dyke Parunak, Robert Savit, and Rick L. Riolo. Agent-based modeling vs. equation-based modeling: A case study and users' guide. In *Proceedings of the First International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pages 10–25, London, UK, UK, 1998. Springer-Verlag.

[84] François Pellegrini. Current challenges for parallel graph partitioning and static mapping. In *Parallel Matrix Algorithms and Applications*, Bâle, Switzerland, June 2010.

[85] Tatjana Petkovic and Sven Loncaric. Supercover plane rasterization - a rasterization algorithm for generating supercover plane inside a cube. In *GRAPP (GM/R)'07*, pages 327–332, 2007.

[86] A. Plastino, C. C. Ribeiro, and N. Rodríguez. Developing spmd applications with load balancing. *Parallel Comput.*, 29:743–766, June 2003.

[87] Saumyadipta Pyne, B.L.S. Prakasa Rao, and S.B. Rao. *Big Data Analytics: Methods and Applications.* Springer Publishing Company, Incorporated, 1st edition, 2016.

[88] Sivasankaran Rajamanickam and Erik G. Boman. *An Evaluation of the Zoltan Parallel Graph and Hypergraph Partitioners.* Feb 2012.

[89] Sivasankaran Rajamanickam and Erik G. Boman. *Parallel Partitioning with Zoltan: Is Hypergraph Partitioning Worth It?.* Sep 2012.

[90] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 25–34, New York, NY, USA, 1987. ACM.

[91] Ferozuddin Riaz and Khidir M. Ali. Applications of graph theory in computer science. In *Proceedings of the 2011 Third International Conference on Computational Intelligence, Communication Systems and Networks*, CICSYN '11, pages 142–145, Washington, DC, USA, 2011. IEEE Computer Society.

[92] Pedro Ribeiro de Andrade. *Game theory and agent-based modelling for the simulation of spatial phenomena.* Instituto Nacional de Pesquisas Espaciais, 2010.

[93] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. High performance cellular level agent-based simulation with flame for the gpu. *Briefings in Bioinformatics*, 11(3):334, 2010.

[94] Claudia Rosas, Anna Sikora, Josep Jorba, Andreu Moreno, and Eduardo César. Improving performance on data-intensive applications using a load balancing methodology based on divisible load theory. *Int. J. Parallel Program.*, 42(1):94–118, February 2014.

[95] Alban Rousset, Bénédicte Herrmann, Christophe Lang, and Laurent Philippe. *A Survey on Parallel and Distributed Multi-Agent Systems*, pages 371–382. Springer International Publishing, Cham, 2014.

[96] Xavier Rubio-Campillo. Pandora: A versatile agent-based modelling platform for social simulation. In *Proceedings of SIMUL 2014, The Sixth International Conference on Advances in System Simulation*, pages 29–34. IARIA Publishing, 2014.

[97] Gudula Runger. Parallel programming models for irregular algorithms. In Karl Heinz Hoffmann and Arnd Meyer, editors, *Parallel Algorithms and Cluster Computing*, volume 52 of *Lecture Notes in Computational Science and Engineering*, pages 3–23. Springer Berlin Heidelberg, 2006.

[98] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[99] Boris Schling. *The Boost C++ Libraries.* XML Press, 2011.

[100] John Scott. *Social network analysis : a handbook.* Sage, London, 2nd ed edition, 2000. Lisäpainokset: Repr. 2005.

[101] Luis Moura E Silva and Rajkumar Buyya. Parallel programming models and paradigms. 1998.

[102] Rishi Pal Singh and Vandana. Article: Application of graph theory in computer science and engineering. *International Journal of Computer Applications*, 104(1):10–13, October 2014. Full text available.

[103] Roberto Solar, Remo Suppi, and Emilio Luque. High performance individual-oriented simulation using complex models. *Procedia Computer Science*, 1(1):447 – 456, 2010.

[104] Roberto Solar, Remo Suppi, and Emilio Luque. High performance distributed cluster-based individual-oriented fish school simulation. *Procedia CS*, 4:76–85, 2011.

[105] Roberto Solar, Remo Suppi, and Emilio Luque. Proximity load balancing for distributed cluster-based individual-oriented fish school simulations. *Procedia Computer Science*, 9(0):328 – 337, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[106] Aad J. Van Der Steen and Jack J. Dongarra. Overview of recent supercomputers, 1996.

[107] Tao Sun, Phil McMinn, Simon Coakley, Mike Holcombe, Rod Smallwood, and Sheila MacNeil. An integrated systems biology approach to understanding the rules of keratinocyte colony formation. *Journal of the Royal Society Interface*, 4(17):1077–1092, 2007.

[108] Tao Sun, Phil McMinn, Mike Holcombe, Rod Smallwood, and Sheila MacNeil. Agent based modelling helps in understanding the rules by which fibroblasts support keratinocyte colony formation. *PLOS ONE*, 3(5):1–17, 05 2008.

[109] Aad J. van der Steen and Jack Dongarra. Handbook of massive data sets. chapter Overview of high performance computers, pages 791–852. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[110] Guillermo Vigueras, Miguel Lozano, and Juan M. Orduña. Workload balancing in distributed crowd simulations: the partitioning method. *J. Supercomput.*, 58(2):261–269, November 2011.

[111] V. Volterra. Variation and fluctuations of the number of individuals of animal species living together. In *Animal Ecology*. McGraw-Hill, 1926.

[112] Ulrike von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[113] Barry Wilkinson and Michael Allen. *Parallel Programming*. Prentice Hall, 1999.

[114] Robin J Wilson. *Introduction to Graph Theory.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[115] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *J. Syst. Archit.*, 49:421–439, November 2003.

[116] Yadong Xu, Wentong Cai, Heiko Aydt, and Michael Lees. Efficient graph-based dynamic load-balancing for parallel large-scale agent-based traffic simulation. In *Proceedings of the 2014 Winter Simulation Conference*, WSC '14, pages 3483–3494, Piscataway, NJ, USA, 2014. IEEE Press.

[117] Andrea Zavanella and Alessandro Milazzo. Predictability of bulk synchronous programs using mpi. In *Proceedings of the 8th Euromicro conference on Parallel and distributed processing*, EURO-PDP'00, pages 118–123, Washington, DC, USA, 1999. IEEE Computer Society.

[118] Dongliang Zhang, Changjun Jiang, and Shu Li. A fast adaptive load balancing method for parallel particle-based simulations. *Simulation Modelling Practice and Theory*, 17(6):1032–1042, 2009.

[119] Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V. Kale. Hierarchical load balancing for Charm++ applications on large supercomputers. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 436–444, Washington, DC, USA, 2010. IEEE Computer Society.