Patterns for multiscale computing
Alowayyed, S.A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Patterns for multiscale computing

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. K.I.J. Maex
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Agnietenkapel
op dinsdag 11 december 2018, te 12:00 uur

door

Saad Abdullah Alowayyed

tegenover te Arar, Saudi Arabia.
The work described in this thesis was carried out in the Computational Science Lab of the University of Amsterdam within the context of the ComPat (www.compat-eu.com) project. The research leading to these results has received funding from King Abdulaziz City for Science and Technology (KACST), Saudi Arabia and from the European Union Horizon 2020 research and innovation programme under grant agreement No 671564.
## Contents

1 Introduction .......................... 1  
   1.1 Multiscale Modeling .......................... 1  
   1.2 Multiscale Modelling and Simulation Framework .......................... 5  
   1.3 Multiscale Computing and High Performance Multiscale Computing .......................... 9  
   1.4 Multiscale Computing Patterns .......................... 10  
   1.5 Thesis Outline .......................... 12  

2 Multiscale Computing Patterns .......................... 13  
   2.1 Computational Challenges at the Exascale .......................... 18  
   2.2 Generic Task Graphs and Multiscale Computing Patterns .......................... 18  
   2.3 Example of an Extreme Scaling Pattern .......................... 21  
   2.4 Discussion and Conclusions .......................... 25  

3 Patterns for High Performance Multiscale Computing .......................... 27  
   3.1 Introduction .......................... 27  
   3.2 Multiscale Computing Patterns and High Performance Multiscale Computing .......................... 29  
   3.3 Design of Multiscale Computing Patterns Software .......................... 31  
   3.4 Applications of the Multiscale Computing Patterns Software .......................... 36  
   3.5 Discussion and Conclusions .......................... 43  

4 Heterogeneous Multiscale Computing .......................... 45  
   4.1 Introduction .......................... 45  
   4.2 Load Balancing in HMC .......................... 47  
   4.3 Red Blood Cell HMM Application .......................... 52  
   4.4 Results .......................... 54  
   4.5 Conclusion .......................... 58  

5 Load Balancing of Parallel Cell-Based Blood-Flow Simulations .......................... 59  
   5.1 Introduction .......................... 59  
   5.2 Parallel Performance Expressed in Terms of Fractional Overheads .......................... 61  
   5.3 Results .......................... 64  
   5.4 Discussion .......................... 69
Science displays its most profound capabilities when it describes how a natural phenomenon work and can predict the outcomes of a future phenomenon before the event takes place. Predicting events in the real world, such as weather patterns, earthquakes, chemical processes and the outcomes of medical interventions, requires one or more high-fidelity simulations in four dimensions (the three spatial dimensions and time), analysis of large quantities of data obtained from the simulations, and the integration and interpretation of the results of the analyses. These computations are among the most demanding in computational science. Not only must the calculations be done rapidly and accurately, but associated uncertainties and error margins must also be determined. The computations are also multiscale in nature, as the event processes must be represented correctly on several spatial and timescales.

Multiscale phenomena are found throughout nature [1–7]. Simulation-based studies of the universe [8], properties of materials [9–13], health and diseases [3, 14–21] and nuclear fusion [22] all focus on processes that interact non-linearly on different spatial and timescales.

This chapter introduces multiscale modelling as it is applied across different research domains. It explains the need for a generic theoretical framework that can be applied universally across disciplines, and provides background on the multiscale modelling and simulation framework (MMSF), which has been developed over the past 10 years to meet this need [23–31]. The chapter then addresses the likely future challenges of implementing multiscale simulations in practice, as high performance computing enters the exascale ($10^{18}$ operations per second) era. It explains how generic computing patterns arising in multiscale applications dictate the scope for novel multiscale algorithms at the exascale. Lastly, it outlines the contribution of this thesis in defining and designing such generic multiscale computing patterns for different categories of multiscale models, and implementing and testing them.

1.1 Multiscale Modeling

We define a multiscale model as a set of coupled single-scale models, where each single-scale model captures a relevant process with definite spatial and timescales, in order to represent a natural phenomenon on multiple spatial and timescales [32].

---

Scales

Scales are used to quantify an object or process. In the context of normal multiscale phenomena in nature, we typically quantify processes using spatio-temporal scales. Our intuitive human spatial scale ranges from a few millimetres to a few metres, and our timescale from a few seconds to the expected maximum human age of about 100 years. Clearly, we need special tools to quantify all processes beyond these scales, and science has successfully explored them all, from the smallest scales of the elementary particles and their interactions to the largest scales of our visible universe.

Usually, spatial and timescales are correlated, especially for multiscale applications in the physics and engineering domains [1]. Macroscopic (macro) models typically capture slow processes occurring over large volumes of space, while microscopic (micro) models capture fast processes occurring within small volumes of space. However, such correlation does not necessarily hold for biomedicine applications [1].

A single phenomenon can also be represented at different scales with different properties. For example, blood flow can be modelled both on a macro scale as a non-Newtonian fluid and on a micro scale as a suspension of blood cells. Moreover, in addition to time and space, more abstract scales can be identified in particular disciplines, such as “leverage growth” in economic models [33]. Generally, to represent a phenomenon, there are no generic scale-selection options, and the best representation depends on the phenomenon itself [1].

Coupling

After defining the single-scale models that will serve as components of the desired multiscale model, we need to “couple” them at different spatio-temporal scales. Coupling submodels means determining when to send information from one single-scale submodel to another, and which information to send. It will also be necessary in many cases to transform the information which is an inherent and non-trivial part of a multiscale model. The question of when to transfer data between submodels depends heavily on the timescale separation between them. The question of which data to send and how to transform it also heavily depends on the submodels and should result in coupling the submodels in physically meaningful ways.

Returning to the example of blood flow, suppose we wish to couple a lattice Boltzmann fluid submodel [34] (a macro-scale model) to one or more fully resolved cell-suspension submodels (micro-scale models) [28]. In the event that we do not have an adequate constitutive equation, a complete run of the micro-scale model(s) could be required to calculate local viscosities for all, or at least most, of the lattice points in the macro-scale model. Then, the macro-scale model uses these viscosities to calculate the fluid velocity and provide the shear rate, density and so on. The lattice Boltzmann blood-flow model represents processes occurring on a larger spatial scale (∼ 1 cm) and a long timescale (∼ 1 s), while the cell-suspension model represents processes occurring on a small spatial scale (∼ 100µm) and a short timescale (∼ 1 ms). This would then lead to a heterogeneous multiscale model for blood flow [31].

Such coupling methods are known as scale-bridging methods – a set of methods to convert data from one scale to another, such as interpolation, projection, sampling and more [1]. Scale-bridging methods are the core of multiscale modelling, and much research has been done on these methods in different fields. Some have been widely adopted by different fields [35], while others are field-specific [36, 37] or problem-specific [38].

1The component single-scale models are called "submodels" henceforth.
Scale-bridging methods must be considered and implemented separately from the submodels themselves, as they must be informed by the (external) spatial and timescales of the overall multiscale model rather than the submodels’ (internal) scales. In other words, submodels are independent of the multiscale model scales, and therefore scale bridging must be implemented outside the submodels using information about the overall scales and knowledge of how to convert data between the individual submodel scales. This is the part of scale bridging that is most prone to error and so requires more verification and validation.

A set of categorised, generic scale-bridging methods would be desirable, by providing a background theory by which to formulate any new model-specific scale-bridging method. The literature contains some notable attempts in this direction, such as introducing Langevin dynamics and dissipative particle dynamics (DPD) to coarse-grained molecular dynamics microsimulations of single particles [39]. However, this is beyond the scope of this thesis.

**Importance of multiscale modelling across the scientific community**

Multiscale modelling is used universally, and research communities in many different scientific fields (e.g., energy, biomedicine, etc.) have had impressive results, e.g., [8, 9]. In most of these fields, progress is determined by our ability to design and implement multiscale models of the particular systems under study [1, 6, 40]. The pan-European Virtual Physiological Human (VPH) initiative\(^2\) is one example, among many, of a widespread community of researchers who consider multiscale modelling as being at the core of their projects [41, 42].

Consistent with this widespread importance, multiscale modelling projects receive a reasonable quota of European Commission funding. Searching the European Commission’s public repository (CORDIS)\(^3\) for projects with the word “multiscale” in their titles or abstracts shows that a total of 558.22 million euros was spent on 250 relevant projects over the past 10 years. More details of the total number of projects and their costs per year are shown in Fig.1.1 (a) and (b).

![Number of Projects](image1.png) ![Cost of Projects](image2.png)

**Figure 1.1:** The total number of European Commission-funded projects with the keyword “multiscale” in their titles or abstracts (a) and their funding (b) per year start from January 1, 2010 to January 8, 2018.

---

\(^2\)http://www.vph-institute.org/

\(^3\)https://cordis.europa.eu/guidance/home_en.html
Notwithstanding the high-level genericity in multiscale modelling, different research communities adopt very different approaches to the design, implementation and execution of their multiscale models [43]. Moreover, these communities prefer different tools. For example, astrophysics and energy researchers tend to use domain-specific tools, while biology and environmental science researchers typically use general-purpose tools [43].

**Importance of a universal theoretical framework for multiscale modelling**

Although researchers have devoted considerable attention to multiscale modelling, no unified framework or agreed terminology for multiscale modelling yet exists. This is impeding communication across research groups and fields, making it difficult for different research groups to co-implement multiscale modelling [1, 23]. For example, some groups refer to serial modelling, whereas others use concurrent modelling. Some of the literature refers to “upscaleing” and “downscaleing” [44], while other papers refer to “coarser” and “finer” models [32, 45]. Even core concepts are expressed in multiple ways. For example, a “submodel” and “multiscale model” in one reference [28] are referred to as a “solver” and “simulation”, respectively, in another [46].

A set of universal theoretical concepts that are interdisciplinary, and neither domain-specific nor application-specific, is needed to facilitate idea sharing, the initiation of new collaborations and interdisciplinary research. However, most of the existing theoretical frameworks are built upon a specific problem, such as predicting tumour growth [47], or in a specific domain, such as energy [48].

With assistance from numerous collaborators, we have developed the multiscale modelling and simulation framework over the past 10 years, to aid the design, programming, implementation and execution of diverse multiscale modelling applications [3, 23–32]. The MMSF has been successful in diverse fields of science and technology, including nuclear fusion [24, 49], computational biology [24, 50, 51], biomedicine [17, 24, 25, 52–58], nanomaterials [9, 13, 24] and hydrology [24]. The MMSF offers numerous advantages, including a clear methodology, common software and algorithms that can be reused across diverse applications, compatibility with both new and legacy programming languages and code, suitability for heterogeneous distributed computing and access to extensive computing resources [32].

Few generic approaches similar to the MMSF are described in the literature. As an example, the U.S. Army Research Laboratory has a multiscale computing approach capability for materials modelling, and a scale-bridging computational framework has been established [35, 59]. The primary purpose of these efforts is to reduce the development and evaluation times and costs of new materials. The framework can connect two different scales (macro and micro) without modification but can be extended to connect three or more hierarchical scales. An important feature of the framework is a standalone evaluation module that couples the macro and micro-scale models. This module performs runs of the micro-scale model, taking care of the scheduling, execution and data exchange activities, on the request of the macro-scale model. Since the exchange mechanism is asynchronous, multiple micro-scale model runs can be performed and managed concurrently by the module.

There are other, less generic frameworks, mainly based on the popular macro–micro coupling method known as the heterogeneous multiscale method (HMM) [60]. Also, Yang and Marquardt [61] have developed a framework that adopts the concepts and the rules of the HMM, such as the connection laws. MMSF, on the contrary, allows different types of coupling taking into account deadlock prevention. In the systematic upscaleing framework [62], the scales of submodels are
identified from the multiscale model, instead of the other way around. MMSF, though, uses the scales of the submodels to derive the scale of the multiscale model.

1.2 Multiscale Modelling and Simulation Framework

The MMSF is a theoretical and practical method for modelling, characterising and simulating multiscale phenomena. As noted above, we have been developing the MMSF over the past 10 years, through the European projects COAST\(^4\), MAPPER\(^5\) and ComPat\(^6\). It has been the basis of a number of publications [1, 23–25, 27, 28, 31, 49, 52, 54, 63, 64] and doctoral theses [25, 65, 66], and in this thesis the latest developments of the MMSF are reported.

MMSF currently comprises a four-stage pipeline from developing a multiscale model to executing the simulation. This process is shown in Fig. 1.2. First, we model a phenomenon by identifying relevant processes that are well described by single-scale submodels (and their relevant scales) using the scale separation map (SSM).

![Conceptual Framework and Computational Framework](image)

**Figure 1.2:** The MMSF pipeline consists of a conceptual part, where the model is specified and coupled (the two panels on the left), and a computational part, where it is implemented and executed (the two panels on the right).

**Scale separation map**

The concept of SSM appears in different frameworks under different names such as “general setting of multiscale model” [47]. Figure 1.3 (a) illustrates the SSM of a macro–micro model.

---

\(^4\)http://www.complex-automata.org/

\(^5\)http://www.mapper-project.eu

\(^6\)http://www.compat-project.eu
In a scale separation map, the submodels are drawn as boxes on a map with characteristic space and time axes, with slow, spatially extended processes (or coarser-grained processes) appearing in the upper-right region, and fast, spatially limited processes (or finer-grained processes) appearing in the bottom-left region. The lines between the component submodels show the coupling, and in total there are five separate ways in which two submodels can interact depending on their location on the SSM. These five interaction regions are scale overlap, timescale separation of the same spatial scale, space scale separation of the same timescale, micro–macro coupling and reversed micro–macro coupling, and are explained in detail in [3, 27]. The concept of the SSM appears in different frameworks under different names, such as the “general setting of multiscale model” used in the multiscale framework for modelling tumour growth [47]. Figure 1.3(a) shows the SSM of a generic macro (M) – micro (μ) multiscale model.

![Scale separation map (SSM)](image)

![Submodel execution loop (SEL)](image)

![Graphical Multiscale Modelling Language (gMML)](image)

**Figure 1.3:** (a) Scale separation map (SSM), (b) submodel execution loop (SEL), and (c) graphical Multiscale Modelling Language (gMML) for a generic macro–micro model.

Note that submodels should not hold information regarding one another; that is, they should be independent. In MMSF, one can describe a submodel as a submodel execution loop (SEL) [28]. Here, we will consider the basic time-driven form of SEL for simplicity and clarity and refer to Borgdorff et al. [23] for general SEL structures.

**Submodel execution loop**

In Fig. 1.3 (b), submodels M and μ have the same structure, but different operations and methods. Solver S is the calculation step, and Boundary B represents the boundary conditions for data exchange from one submodel to another. Observations of the system, which comprise the submodel
output, are categorised as either intermediate \(O_i\) or final \(O_f\) observations. The initialisation step of the system is represented by \(f_{init}\) \[23, 28, 67\].

**Coupling template**

Next, we define the mutual couplings between single-scale submodels. These couplings, which define any multiscale model, are clearly application dependent and are themselves not addressed by the MMSF. However, the MMSF provides a set of abstract coupling templates to use as a starting point or to be used as programming templates for actual implementation of a multiscale model, as is done e.g., in Muscle-HPC \[68\]. In Fig. 1.3 we show one example of a coupling template for the call–release sequence between submodels. For more in-depth information, see Borgdorff et al. \[23\].

When executing a multiscale simulation, two properties of coupling are of primary concern in the MMSF, namely the number of submodel instantiations and the number of synchronisations between submodels. The synchronisations can be one-off acyclic couplings, a fixed number of cyclic couplings or a dynamic number of cyclic couplings. The same categorisation is applied for the number of submodels, one-to-one couplings, one-to-many couplings or one-to-dynamic number of submodels \[23\]. So far, most realisations of the MMSF have assumed at best static instantiations and synchronisations. In this thesis we describe first attempts to allow for dynamic instantiations, when realising implementations of the hierarchical multiscale method.

**Multiscale Modelling Language**

The overall multiscale model, i.e., the single-scale submodels and their mutual couplings, is specified with the Multiscale Modelling Language (MML) \[23, 29\], thereby forming the architecture of the multiscale model. MML is used to describe the scales and computational requirements of submodels and any scale-bridging components needed. MML provides a generic and well-defined way to communicate the structure of a multiscale model.

MML is represented in the form of computer-readable text files (the xMML) or graphical representations (gMML) \[25\]. xMML uses xml to express the coupling between submodels, also specifying details of the SEL coupling templates and the spatio-temporal scales. gMML is the graphical representation of MML using task graphs, somewhat similar to UML. The latter is a graphical and human-friendly way to represent submodel coupling, which facilitates collaboration and communication. Figure 1.3(c) shows the gMML representation of processes M and \(\mu\). Note that all symbols used in gMML have a clear meaning, see \[23, 32\].

Although the details of the actual coupling, so the scale-bridging model, is a core part of any scientifically meaningful multiscale model (see e.g., \[6, 7\]), the MMSF provides a generic framework for multiscale modelling and simulation once the submodels and scale-bridging methods are available \[24, 32\].

**Coupling toolkits**

After defining a multiscale model theoretically, we need to implement it such that the multiscale simulation can be executed as efficiently as possible on available computing resources. This is not an easy task and often fails at first, especially when the coupling is hard-coded, which may be problematic for subsequent follow-up projects. Thus, the main objective for the coupling toolkits or coupling libraries is to couple implementations of submodels via scale-bridging algorithms in
a generic way, with the fewest possible modifications of existing submodel code (i.e., using the execution environment). By "generic", we mean compatible with the different methods that submodels may utilise (mesh-based, agent-based, etc.) and the different programming languages in which they may be written in (C++, Fortran, Java, Python, etc.). The actual coupling interface is an important decision. For example, Message Passing Interface (MPI) is fast but only works for certain programming languages, while a file-based medium is slow but does not rely on any specific programming language. Furthermore, the coupling toolkit or library should not increase the simulation overhead.

A coupling library such as the MUtliScale Coupling Library and Environment 2 (MUSCLE2) [25] ensures that communication between heterogeneous components is possible, with minimal, localised changes to the single-scale submodel code. MUSCLE2 supports multiple programming languages, which makes it a perfect tool for legacy code. Also, the configuration file is straightforward and adheres to the MMSF rules [25].

Another tool, or interface, is the Multiscale Universal Interface (MUI) [46]. MUI is a C++ MPI coupling tool. However, it does not allow for abstract time separation; instead, the frequency of sends depends on the push–fetch feature integrated into the scale. This, from the modulation point of view, will not obey the separation of concerns. Also, this will add additional work on the development side. However, when automated, such a feature could be quite beneficial, e.g., in smart conduits in MUSCLE2.

For concurrent submodel runs, load imbalance and/or system overwrite could occur. MUI solves this problem using "time frames", where virtual containers hold the system for each time step, when it is not clear whether these time frames consume memory. Finally, unlike MUSCLE2, MUI has no clear theoretical framework.

In MUSCLE2, the communication between submodels is implemented using conduits to transfer data, and filters to transform data (i.e., to perform the scale bridging). The two analogous concepts in MUI are "Data points" and "Data samplers", but the main difference is that the data points method goes into too much detail about the multiscale application by assuming a virtual container to hold the system, and treating the system as a cloud of points.

MUSCLE2 and MUI are similar in that they separate scale bridging from the submodels. Although MUI has a number of interpolation engines, the user can define a custom interpolation functions the same way as filters in MUSCLE2. Additionally, both are easy to use with reasonable overhead [25, 46].

The common component architecture (CCA) [69] is a more flexible coupling tool than MUI, in the sense that the solver decides the content of communication, as MUSCLE2 does. However, CCA assumes only shared memory on a single processor, and it requires more code refactoring than MUSCLE2 and MUI.

The main dilemma in choosing a coupling tool is to decide whether it is better to have a fast coupling tool or whether it may be more important to choose a tool that needs less code refactoring. For example, Kratos [70] is a fast, light and multi-disciplinary coupling tool, but has a strict number of classes of solvers and requires significant code refactoring.

More specific examples of coupling tools include MacroMicroCoupling (MaMiCo) [71] for macro–micro coupling, the domain-specific AMUSE [72] for astrophysics, the Model Coupling Toolkit (MCT) [73] for multiphysics applications and the method-specific Data Driven Environment for Multiphysics Applications (DDEMA) [74] for mesh-based PDE solvers.

A new MUSCLE implementation, MUSCLE-HPC [68], has faster MPI communication than MUSCLE2. However, it strictly imposes the structure of SELs for the submodels, which requires
code refactoring. Moreover, because it uses MPI-2.2, only a limited number of programming languages, such as C++, Python and Fortran, can be integrated.

Execution of multiscale applications

In the last step of the pipeline, submodels are executed on suitable computing infrastructure. Each submodel may require different computing resources. Some may be massively parallel and/or may require special hardware or software. In MMSF, the submodels can be distributed across several computers without additional software development [24, 26, 53, 64].

Once the multiscale model is running, its results need to be verified. Errors can be generated from discretising the space domain or from the coupling method, or they can be propagated from one submodel to another. A large amount of research is being conducted on uncertainty quantification [75, 76], multiscale modelling validation [77, 78] and error analysis [79, 80]. Without going into too many details, it may be possible to determine multiscale model errors from knowledge of the submodel errors, and to integrate such error analysis into the framework. Some trials are underway to map the "distance" between submodel errors and overall errors [30, 81].

In general, the minimal acceptance criterion for a multiscale model is:

\[
\frac{\text{Cost of multiscale model}}{\text{Cost of fine-scale submodel}} \ll 1,
\]

with a reasonable error [28]. Reducing this cost may require more and faster computing resources.

1.3 Multiscale Computing and High Performance Multiscale Computing

The complexity of multiscale phenomena restricts our capability to perform high-fidelity simulations that can predict the behaviour of a given phenomenon accurately in all situations. Capturing a small number of coupled processes in a multiscale simulation may easily reach the current limit of high performance computing (HPC) of \(10^{15}\) operations per second (the petascale).

As an example, a composite nanomaterial can be simulated using millions of atoms. The atoms’ interaction potentials depend on their surrounding electron structures, and accurate simulations need to model this for millions of individual particles, although the simulations are necessarily coarser-grained in practice. Such simulations have already resulted in insights into chemically specific structural self-assembly and the prediction of large-scale material properties [9]. However, these simulations are still limited to the micrometre scale, with the prediction of material performance in applications such as the automotive and aerospace industries remaining unachievable. Stepping up to that scale will require simulating billions of particles, as well as managing the non-linear increase in computational load as the simulation timescale is increased. It would also require advanced statistical analysis, which itself imposes further demands on computational resources. It is clear that performing atom-level simulations to inform coarse-grained materials simulations — and eventually also, for example, for finite element calculations — would require resources beyond the petascale computing power currently available.

To move from the current situation to more realistic macroscopic computational regimes will require new approaches in multiscale computing to extend discretised approaches to the
continuum level, requiring a 100- to 1000-fold increase in computing capabilities from what we have now.

The importance of multiscale modelling in many domains of science and engineering is still increasing, and is clearly demonstrated in e.g., [1, 7, 42, 43, 69]. Therefore, we must anticipate that multiscale simulations will become an increasingly important application of future HPC resources, necessitating the development of sustainable and reusable solutions; that is, generic algorithms for multiscale computing. As we move into the exascale \(10^{18}\) operations per second era, we need to drastically change the way we use HPC for simulation-based sciences [82].

Strong computational considerations also dictate a need to shift the paradigm for usage of high performance computers from the conventional promotion of monolithic code, which scales to the full production partition of these computers, to much more flexible computing patterns. This calls for new algorithmic approaches such as the ones proposed in this thesis.

Computational scientists have developed numerous effective ways to perform spatial domain decomposition. However, petascale and future exascale machines can only reach these performance levels by aggregating a large number of cores whose individual clock speeds are no longer increasing. As a result, these high performance computers are becoming "fatter", not faster, and speed-up is only achievable by efficient parallelism over all the cores.

However, because the parallelism is usually applied to the spatial domain, we are increasingly simulating larger slabs of matter, applying weak scaling by using more particles, a higher grid resolution or more finite elements. Yet, it is often the temporal behaviour that one is really interested in, and that behaviour is not extended by adopting larger computers of this nature, or by modelling a larger region of space. Moreover, since the timescales of scientific problems of interest usually scale as a non-linear function of the spatial volume of the system under investigation, each increase in the duration of time we model effectively also requires more wall clock time just for simulating a larger physical system.

We are getting no closer to studying large-space and long-duration behaviour with monolithic codes. Accelerators (such as GPUs) and special-purpose architectures [83–85] can speed-up many floating point calculations in particular cases, such as molecular dynamics, often by a factor of up to 10, but this is not sufficient to bridge the vast range of timescales of concern, from femtoseconds to seconds, hours and years, nor indeed to quantify the uncertainty in today's still all-too-prevalent "one-off" simulations.

What is needed are more innovative ways of bridging this divide. Multiscale computing (MC), as we propose it, is able to do this by deploying its single-scale component submodels across heterogeneous architectures, mapped so as to produce optimal performance and designed to bridge both time and spatial scales. Thus, we have embarked on a programme to efficiently deploy componentised multiscale code on current and future high performance computers, and thereby to establish a new and more effective paradigm for exploiting HPC resources.

Indeed, advanced multiscale algorithms, in combination with exascale resources, will help us transition to predictive multiscale science. To make this possible, we will need generic MC algorithms capable of producing high-fidelity scientific results and which are scalable to emerging exascale computing systems. We call this high performance multiscale computing (HPMC).

1.4 Multiscale Computing Patterns

Using our basic definition of a multiscale model (as a collection of coupled single-scale submodels), we can identify generic computing patterns arising in multiscale applications that dictate the
MULTISCALE COMPUTING PATTERNS

scope for novel multiscale algorithms at the exascale. We call these multiscale computing patterns (MCPs). This thesis will introduce MCPs and propose a classification of these computing patterns, implement the vision and concepts and develop MCP software in order to prepare and facilitate multiscale applications at the exascale.

Challenges faced by exascale computing include scheduling, the robustness of algorithms and their implementation on millions of processors, data storage and input/output for extreme parallelism, fault-tolerance and reducing energy consumption [86–89]. New and unique algorithms are needed across the software spectrum, bridging between the applications and the hardware environments. The algorithms need to be designed specifically to address the expected exascale challenges. We believe that, drawing on the concept of generic MCPs, we can realise a separation of concerns, where the challenges stated above can be resolved to a large extent on the level of the MCPs, while the multiscale application developers can focus on composing their multiscale simulations. This would result in shorter development cycles for multiscale simulations and much more reliable multiscale computing on exascale machines.

Ultimately, MCP software aims to allocate resources optimally to execute the multiscale simulation. However, the optimal resources remain to be defined. This is user dependent, but the result cannot be other than the least makespan time, the least time to obtain results, the least power consumption, and/or runs that fit the budget. Patterns can take all these into consideration, except the budget, which varies from one system to another. Determining the least makespan time depends on the scalability of the multiscale application, which is not trivial to determine because it depends on the scalability of the submodels, communication between submodels, and the type of chosen computing infrastructure to run the multiscale application. The time to obtain results is more complicated because, in addition to the makespan time, we need to include queuing time. Queuing times depend on the availability of computing nodes and vary from one supercomputer to another. Power usage concerns fewer users at present, but power is predicted to become more important in the future [90]. It is even expected that the budget of supercomputers will be calculated in joules per hour instead of cores per hour. Moreover, Green500\(^7\), which is an equivalent concept to Top500\(^8\), is starting to gain more attention. Like time calculations in multiscale applications, power consumption is far from trivial.

The MMSF is a natural starting point for developing generic algorithms for multiscale computing on HPC resources, in terms of both its theoretical approach to multiscale modelling and practical aspects for multiscale simulations. The MMSF is the foundation for realising and implementing MCPs that can lead to an exascale multiscale simulation framework. This requires transforming the current MMSF approach for a new kind of HPC, HPMC.

To optimise the efficiency of a multiscale application, and to understand various components of patterns, we need to look at the performance figures of the single-scale submodels. In this thesis, as a representative example, we analyse and optimise the performance of our suspension simulation model framework, HemoCell\(^9\) [91], which has been developed and optimised over four years. We also consider another source of efficiency loss in these types of applications, namely the load imbalance between processes due to non-homogeneous distributions of suspended cells. Thus, we formulate a model for the fractional load-imbalance overhead, validate it using HemoCell, compare the load-imbalance overhead with the communication overhead and develop a load balance library to decrease this emerging fractional overhead and so gain a speed-up.

\(^7\)https://www.top500.org/green500/
\(^8\)https://www.top500.org/
\(^9\)https://www.hemocell.eu/
1.5 Thesis Outline

This thesis first defines MCPs for different categories of multiscale models and outlines the vision for implementing these MCPs in Chapter 2. Chapter 3 is mainly dedicated to the design of these patterns and development of the MCP software for two types of patterns: extreme scaling (ES) and replica computing (RC). The third pattern, heterogeneous multiscale computing (HMC), is discussed in detail in Chapter 4. In Chapters 5 and 6, a model for the fractional load-imbalance overhead is formulated, and a load balance library based on this model is illustrated, as well as some performance figures concerning the cell-suspension model under investigation (HemoCell). Finally, Chapter 7 draws conclusions and makes suggestions for future work.
Abstract

We expect that multiscale simulations will be one of the main high performance computing workloads in the exascale era. We propose multiscale computing patterns as a generic vehicle to realise load balanced, fault-tolerant and energy-aware high performance multiscale computing. Multiscale computing patterns should lead to a separation of concerns, whereby application developers can compose multiscale models and execute multiscale simulations, while pattern software realises optimised, fault-tolerant and energy-aware multiscale computing. We introduce three multiscale computing patterns, present an example of the extreme scaling pattern, and discuss our vision of how this may shape multiscale computing in the exascale era.

An important ingredient of MMSF that so far has not been fully exploited is the notion of task graphs for multiscale computing. A task graph is a directed acyclic graph of tasks (the nodes) and their dependencies or data flows (the edges). It can be used for scheduling on parallel and/or distributed computing resources [92]. It can also be seen as a serialised or unfolded graph of the MML description, which may be cyclic. Borgdorff et al. introduced task graphs when specifying the foundations of the MMSF [23], primarily for purposes of deadlock detection, validity checking, and for estimating computational costs and scheduling. As shown in MMSF, a task graph can be derived from an xMML specification of a multiscale application, which in turn can be used as input for scheduling software. We have shown that task graphs can automatically be derived from xMML [23] and demonstrated the use of task graphs for one specific application [26]. These task graphs will perform an important role in future multiscale applications, since they can be used to create Multiscale Computing Patterns.

We present an example of a task graph in Fig.2.1. This relates to a multiscale model for in-stent restenosis, an unwanted response of coronary arteries after balloon angioplasty and stenting [24, 26, 57, 93, 94]. The ‘SMC’ refers to a slow dynamical process, the response of smooth muscle cells in the coronary arterial wall, with a typical timescale of days. Drug diffusion (DD) simulates diffusion of inhibitory drugs from a stent into the arterial wall and has a timescale of

---

minutes, and the blood flow (BF) has a timescale of one heartbeat, so a second. The growth of SMCs is dictated by their internal biology and at the same time is regulated by the blood flow and inhibited by high enough drug concentrations. Given the timescale separation, the slow dynamics (of SMC) is in quasi-equilibrium with the fast dynamics (of DD and BF), which are independent from each other and can be simulated in parallel after each iteration of the slow SMC dynamics. This is clearly expressed by the task graph.

Our central assumption is that one can identify generic computing patterns that allow developing algorithms for common scenarios of multiscale computing that cover the vast majority of multiscale scientific applications. We call these multiscale computing patterns. Their main benefit lies in hiding the intricacies of the efficient usage of exascale machines — in terms of extreme parallelism and related load balancing, energy awareness and fault tolerance — from the multiscale application developers, which should lead to much shorter development cycles and at the same time much more robust and efficient multiscale simulations.

In general, computing patterns can be identified on many levels. They exist deep within the computational kernels of complex application codes (e.g., matrix manipulations relying on BLAS routines, as implemented, for example, in the linear algebra package LAPACK; see, e.g., [95]) as well as in call sequences involving higher-level subroutines and multiple component codes (such as the common component architecture; see e.g., [96]). Since we have already established that there are many generic aspects to multiscale simulations [23, 32], exploiting additional genericity is central to the approach we adopt. Indeed, we look for computing patterns at a higher level, namely call sequences of individual single-scale submodels comprising the components of the multiscale model. This leads to only a few multiscale computing patterns.

We define multiscale computing patterns as high-level call sequences that exploit the functional decomposition of multiscale models in terms of single-scale models. The task graph
introduced in the previous section will be the main ingredient for expressing MCP in terms of objects within the MMSF.

We anticipate a number of key challenges when using today’s multi-petascale platforms and future exascale resources in relation to developing and implementing the MCPs. In particular, unprecedented costs in exascale computing have multiple aspects, as running applications on such scales is accompanied by an exceptional cost requirement in terms of computer time (measured, e.g., in core or node hours), energy consumption, monetary cost, or time to completion (particularly in urgent computing). Throughout this section, we will discuss cost from a generalised perspective, and describe parts in each of the MCPs that are particularly cost critical. Each MCP has different cost-critical parts, and these differences help to more clearly demarcate the definitions for each of the patterns.

Based on our previous experience with multiscale modelling, we have identified three computing patterns that we believe are the most pertinent for High Performance Multiscale Computing:

• Extreme scaling (ES) is where one (or a few) single-scale model requires exascale performance; this is coupled to other, less-costly, single-scale models.

• Heterogeneous multiscale computing (HMC) is where a very large number of micro-scale models are coupled to a macro-scale model.

• Replica computing (RC) is where a large number of copies (replicas) are executed, which may or may not exchange information. Replica computing is plainly — in “vanilla” form — a single-scale pattern, but it can also be regarded as a degenerate form of multiscale computing, in which the models are all at the same scale. There are both incremental variants such as replica-exchange, and the extension to true multiscale modelling which can be based on this form; the use of replicas at any level of computing provides an important handle on uncertainty quantification.

These three patterns, and combinations thereof, cover a broad range of possible call sequences in high performance multiscale computing. It may turn out that other MCPs should also be considered in due course. These MCPs offer a generic framework for efficient multiscale computing in the emerging exascale performance era, and the three MCPs that we introduce here will prove to be important in that context.

It is important to realise that the MCPs, while being prepared for the challenges listed above, primarily catch the computational structure of the multiscale models, independent of the details of the underlying exascale machines. The latter is hidden in the tools, services and middleware, on which the multiscale computing patterns will rely. This separation of concerns is a main goal of the ComPat project\(^1\), where we are actually implemented the ideas discussed in this chapter.

The extreme scaling computing pattern represents a specific class of multiscale applications where one (or perhaps a few) of the single-scale models in the overall multiscale model by far dominates all others, in terms of computational and/or energy cost. Such a dominating primary model is expected to scale to very large systems (i.e., multi-petascale or above), and the efficiency of the primary model largely determines the efficiency of the full application. Consequently, one of our goals is to ensure minimal interference by the other single-scale models, so-called auxiliary models. These typically have a much lower computational and/or energy cost and might even be sequential codes. Load-balancing, decentralised communication, and computation overlapping

\(^1\)http://www.compat-project.eu/
are some of the techniques we can use here, depending on the relation between the primary and auxiliary models.

The extreme scaling pattern applies when, for instance, in addition to the time and space scale differences, one submodel requires a strong increase in dimensionality or resolution compared to the others, and therefore becomes cost critical. The cost difference between the primary model and the auxiliary models introduces a set of load balancing challenges and possible bottlenecks. The challenge is to efficiently execute the primary model as it is coupled to the auxiliary models, minimising additional overheads incurred by the coupling and load imbalance, and to propose new variations of the pattern, which include mechanisms for fault-tolerant, energy-efficient, and data-intensive computing.

In the heterogeneous multiscale computing pattern, we couple a macroscopic model to a large and dynamically varying number of microscopic models. The pattern is based on the heterogeneous multiscale method (HMM) [2, 60], which involves a class of modelling approaches wherein the constitutive equations of a local state in the macro-scale system of interest are not known in closed form, mainly as a consequence of the complexity of the processes at the micro-scale. The basic philosophy of HMM is to apply a numerical solver to the macro-scale equations and to provide the missing macro-scale data using an appropriate micro-scale model. HMMs hold the promise to simulate very complex phenomena, directly coupling detailed microscopic dynamics to emerging macroscopic behaviour. Instead of relying on heuristic constitutive relations or on idealised theory containing simplifying assumptions, HMM allows all microscopic details to be taken into account while at the same time being able to simulate macro-scale dynamics. As examples we can mention HMM models for suspension flow [65] or for galaxy merger simulations [97]. HMM-type high performance computational frameworks are still rare; we seek to establish one by defining a multiscale computing pattern for HMM, which is HMC.

The number of micro-scale models required in HMC depends on the spatial properties of the macro-scale model, and can in some cases easily be in the order of 10⁶ or more (e.g., for the suspension flow, where a micro-scale simulation is required for each lattice point in the macro-scale simulation). In addition, each micro-scale model can be a detailed 3-D simulation requiring substantial parallel computing resources (e.g., a cell-based blood-flow model, explicitly taking individual red blood cells into account [98]). The large number and size of the micro-scale models causes them to dominate the computational and energy cost of the application as a whole, and they are therefore cost critical. Consequently, it is essential that these models are efficiently mapped and executed on (exascale) HPC resources, and that any overheads incurred by interactions with other components are minimised. Additionally, the cost of micro-scale models in HMC, especially when mapped to exascale resources, may result in unprecedented data challenges, which can be addressed by introducing a scalable data management software architecture.

As part of this architecture, an on-the-fly database is needed to limit the number of required micro-scale simulations [31, 99]. This database serves to store previously computed data and, where desirable, interpolates between already computed values to provide input to the macro-scale model. This is feasible because the amount of data passed up to the macro-scale model is usually not large, perhaps a few floating-point numbers representing quantities of interest (such as the viscosity for fluid problems) for each micro-scale model.

The on-the-fly database will be managed by an HMC manager. Here, the coarse-scale model first sends a request to an HMC manager for the properties that it requires more information. The manager then consults the database for the needed information. A user-defined algorithm will decide whether the cached information is sufficiently accurate, and whether interpolated
values may be used. If the available information is insufficient, the manager will start a new micro-scale simulation to get more accurate fine-scale information. For simulations where the evaluation of cached results is computationally intensive, this evaluation will be delegated to a separate computing resource in order not to overload the HMC manager. By using asynchronous I/O and offloading intensive calculations, the HMC manager is able to handle many requests simultaneously.

*Replica computing* is a multiscale computing pattern that combines a potentially very large number of terascale and petascale simulations (also known as ‘replicas’) to produce scientifically important and statistically robust outcomes. The replicas are not part of a larger spatial structure (as is the case in, for example, HMC), but they are applied to explore a system under a broad range of conditions. A good example of replica computing is the Binding Affinity Calculator, a quite involved workflow in which production-scale molecular dynamics is performed to compute binding affinities between small compounds and proteins, typically followed by similar computation-intensive processing of the trajectory data accumulated in that stage [100–102]. Another example is daily weather forecasting, where many replicas are executed in order to assess uncertainties in the predictions [103, 104]. Replica computing is set-up through an initialisation stage, which determines the simulations required to explore or incorporate a given parameter space. This initialisation is then followed by one or more sequences of simulation and data processing. In general, within replica computing we distinguish three scenarios:

1. **Ensemble simulations.** Here a large set of models (replicas) is initiated, run and analysed. The results of these models, which can either be large-scale computing jobs themselves or require small amounts of computing resources, are then provided as part of the initial conditions of one or more secondary models. These models may operate on a much larger space- and timescale and in that case are usually critical to the overall application in terms of computational and energy cost, or may need just a fraction of resources as compared to the replica computations.

2. **Dynamic ensemble simulations.** Similar to ensemble simulations, here the approach relies on a set of small-scale models that are initiated, run, and analysed. However, in dynamic ensemble simulations, the analysed data is used to rerun a new, better-calibrated or complementary set of small-scale models, allowing system behaviour to be characterised in complicated parameter landscapes. This landscape in itself can be multiscale. The results of these model executions are then used as a basis for one or more secondary models, similar to the ensemble simulation scenario.

3. **Replica-exchange simulations.** This scenario introduces a set of models with varying parameters (e.g., temperature or spatial characteristics), which are run concurrently. Simulation data is exchanged between these single-scale models at runtime, for example to allow individual particles to migrate from one model to another. These exchanging replicas may be multiscale in their own right, or they can provide statistically robust data, which is then used for more coarse-grained models, that operate on larger time and length scales. Replica-exchange simulations are already used in a variety of fields on smaller scales, including in, e.g., materials science [105], climate sciences [106], biomedicine [107] and origin-of-life studies [108].
2.1 Computational Challenges at the Exascale

As exascale architectures are being designed and developed, it has become obvious that these new architectures will introduce a range of new computational challenges. Today’s largest (and third-most efficient) supercomputer, TaihuLight, relies on low-power compute units at extreme parallelism ($\approx 11$ million cores), accompanied by limited memory capacity ($0.1$ GB/core, whereas $> 1$ GB/core used to be commonplace before). In general, the TOP500 list shows a clear trend towards heterogeneous systems, and we expect that exascale systems will present us with significant non-uniformity of performance and reliability. Here, we elaborate on key exascale challenges and describe how MCPs can help to address them efficiently. Our objective is to address all of these exascale challenges as far as possible on the generic level of the MCPs, as it will allow us to formulate unified solutions for these challenges across a class of applications.

The extreme parallelism presented by the massive core counts implies a need for advanced load balancing and scheduling to maximally exploit the offered compute power. The use of MCPs instead of monolithic applications already helps to accommodate this, but we additionally need to develop scheduling and load-balancing algorithms that account for the requirements of each respective MCP in terms of compute efficiency, memory efficiency, and energy efficiency. Constrained optimisation approaches can help us to introduce pattern-level optimisations, incorporating awareness of other algorithmic components of the multiscale simulation framework and identifying common scheduling and load-balancing trade-offs for MCPs.

Failure rates tend to scale linearly with the number of cores, and are expected to be commonplace for models using extreme parallelism. We therefore require fault-tolerant software algorithms, ideally defined on the generic level of MCPs, that are capable of handling hardware failures by applying heuristics for fault recovery, satisfying the need for recomputation, or accounting for missing or incomplete data (e.g., using statistical techniques such as data imputation or maximum likelihood). The heterogeneity of the architecture may also lead to non-uniform reliability, and scheduling algorithms could support the deployment of the cost-critical components in our MCPs on resources with a higher reliability.

A main constraint for exascale systems will be power consumption. We therefore need to take energy efficiency into account throughout the whole development process of the multiscale computing algorithms, and deploy multiscale computing applications such that we achieve the best trade-off between performance and power consumption [109]. To accomplish this, we should introduce energy efficiency as a constraint parameter in global optimisation algorithms, and identify regimes where MCPs can take advantage of hardware capabilities such as reducing clock frequency or switching cores on/off dynamically.

2.2 Generic Task Graphs and Multiscale Computing Patterns

An important next step, as we will argue, is the realisation that the MCPs can, in some form, be expressed on the level of the task graph. The task graph, as explained above, is a directed acyclic graph used to determine the execution order of submodels, to schedule submodel dependencies, and to estimate runtime and communication cost. The key idea is that we define generic task graphs for each MCP, such that application-specific task graphs can be embedded in the generic task graphs. This embedding should be automated. Next, we use the generic task graph to obtain an optimised mapping of the application to an HPC resource, and try to find generic algorithms for this. What exactly is meant by an ‘optimal’ mapping needs to be defined, or can be
made application specific. In any case, it should be optimised with respect to several dimensions (efficient use of resources, power consumption, wall clock time, load balancing, fault tolerance). The way to proceed is that for each generic task graph we need to specify sets of optimal execution profiles, or define constrained optimisation problems that should be easily solvable when fed with details of the specific applications. Figure 2.2 summarises the approach. An MCP is a tuple of a generic task graph plus data or models on the performance of single-scale models (left in Fig. 2.2), a specification of a specific multiscale application in terms of the MMSF (right part of MMSF), and a set of algorithms and heuristics that combine this into detailed input/configuration files for the execution environment in which the multiscale simulation will be executed.

![Figure 2.2](image)

**Figure 2.2:** Multiscale computing patterns implemented as generic task graphs and algorithms to generate sufficient information for the execution engines.

To summarise, the approach that we follow will be that a task graph is generated from the application-specific xMML description. This is then taken together with execution recipes specific for an MCP and performance models, or data, for the single-scale models, to determine the actual execution profile. This will finally be specified in configuration files for the execution middleware. This immediately leads to a number of key questions:

1. Can we find such generic task graphs for each pattern?
2. How can we map specific applications to such task graphs?
3. How can we use the generic task graphs to set-up optimised executions on HPC machines?
4. What information is needed and on which level should we make this work?

In the remainder of this chapter, we formulate some partial answers and work out one example. In future work we intend to generalise these findings to create fully fledged MCPs.
We have worked out generic task graphs for the ES, the HMC, and RC patterns, see Figs. 2.3, 2.4 and 2.5. The meaning of the boxes and symbols in these figures is shown in the legend in Fig. 2.6.

**Figure 2.3:** Generic task graph for ES. The version on the left shows the graph with both serial and parallel auxiliary models, the version on the right only has a serial auxiliary model.

**Figure 2.4:** Generic task graphs for HMC.

**Figure 2.5:** Generic task graphs for RC. The version on the left shows the static variant for ensemble simulations and replica-exchange, while the version on the right is for dynamic ensemble simulation.

Figure 2.3 (left) shows the generic task graph for ES, where a collection of auxiliary models can either be executed in parallel with the primary model, or in series with the primary model. Figure 2.3 (right) shows the version where only a serial auxiliary model is present. It should be understood that the graphs are repeating elements in the overall task graph. So, considering the task graph for the In-Stent Restenosis application in Fig. 2.1, and knowing that the fluid flow
process is cost critical, we can identify the BF (blood flow) process with A in the generic task graph, the DD (drug diffusion) process with the parallel auxiliary model $B_p$ in the generic task graph, and SMC (smooth muscle cells) iterations together as the serial auxiliary model $B_s$ in the generic task graph. We find many repeating units of the generic task graph for ES in the task graph from Fig. 2.1, typically a few thousand. We also find that the application in Fig. 2.1 has an initialisation phase as well as a post-processing phase (not shown in Fig. 2.1), which are currently not captured by the generic task graphs.

Depending on the execution behaviour of the primary and auxiliary models on HPC machines, a specific execution of the ES graph is considered. The main aim is to align the auxiliary models to produce their data before the primary model requires it. This would be achieved by either pre-computing the values of non-scaling models or interleaving simulations as suggested by the performance model, see also the discussion in the next section.

For HMC, Fig. 2.4, a large and dynamic number of micro-scale simulations is coupled with one macro-scale model, with a HMM database in between. The role of the database is to prevent computing of previously computed results, to interpolate between earlier computed results, and to submit micro-scale simulation jobs when needed. The latter could be done in a pro-active way, if resources allow it, to start pre-computing quantities in anticipation of requests from the macro-scale solver. A specific execution graph for this pattern depends on the execution behaviour of the macro- and micro-scale models on HPC machines.

For RC, see Fig. 2.5; we find two variants, which capture the behaviour of the three types of replica computing that we defined. In both cases a potentially large set of replicas $A_1^+$ are executed independently and then feed into a second master process $A_2$. In RC, if a replica fails, a restart is not immediately needed, as long as the overall statistical quality of the ensemble that is computed by the RC application is maintained. This is a main distinction with HMM, where if a micro-scale simulation fails it must be restarted, as the database requested output from this micro-scale simulation.

2.3 Example of an Extreme Scaling Pattern

A very common example of a multiscale model is a multi-domain scenario [23], where in one part of a computational domain a micro-scale simulation is coupled to a macro-scale simulation in another part of the computational domain. This is typical in situations where one needs to
zoom into molecular or even quantum-mechanical details in small regions of space, as e.g., in the classical example of multiscale modelling of crack propagation in materials [110]. Although the micro-scale simulation will usually take up only a very small portion of the computational domain, the required computational resources for the micro-scale simulations are orders of magnitude larger than for the macro-scale simulation. This immediately puts this scenario into the extreme scaling MCP, where the micro-scale simulation is the primary model, and the macro-scale simulation is the auxiliary model. The application-specific scale bridging dictates whether the macro-scale simulations are serial or parallel auxiliary models. Regardless of these details, an important question is how the available computational resources should be distributed over the micro-scale and macro-scale simulations (as already briefly discussed in the pioneering work by Broughton et al. [110]), how fault tolerance should be realised, and how such simulations could be made energy aware. A first example of how this may be realised on the level of the MCP is discussed below.

One of our own applications is in the field of computational biomedicine [53]. We are in the process of developing the “Virtual Artery” [54], where cell-based (micro-scale) models are coupled to continuous tissue (macro-scale) models in a multi-domain scenario. As an example, we consider the case of a cell-based blood-suspension model as the primary model, coupled to continuous blood-flow models as the auxiliary models, see Fig. 2.7. The idea would be that in an artery, where blood flow is modelled by a continuous flow solver, there are small regions where we actually want to model the behaviour of the red blood cells and platelets. This could be in an aneurysm, for example, to better understand possible thrombosis inside the aneurysm cavity [111].

![Figure 2.7: Example of an ES application, a suspension simulation (the primary model) coupled to two continuous blood-flow models (auxiliary models) that provide the in- and outflow conditions for the suspension model.](image)

The resulting scale separation map, gMML and task graph are shown in Fig. 2.8. Note that we assume two instantiations of the auxiliary model representing the inlet and outlet regions of the suspension flow domain. Also note that the embedding of this application task graph into the generic ES task graph (Fig. 2.3, left) is straightforward; the two instantiations of the continuous flow solver together form the serial auxiliary model, and the parallel auxiliary model is empty.

We formulate a generic and straightforward performance model for this ES MCP. We assume that we have all information on the performance of the single-scale models that make up the multiscale application. This is in line with the MMSF philosophy, whereby the single-scale models and the scale-bridging algorithms are available and implemented on available resources. The MMSF provides the means to compose the full multiscale model and execute it in an efficient way on available resources.

Call $T_{pr}(P, N_{pr})$ the execution time of the primary model as a function of the number of processors $P$ and the problem size of the primary model $N_{pr}$. Likewise, call $T_{aux}(P, N_{aux})$ the
Figure 2.8: The SSM (left), gMML (middle), and task graph (right) for the Extreme Scaling multiscale computing pattern coupling a primary model for blood-suspensions to two instantiations of the auxiliary model.

The execution time for the serial auxiliary model as a function of $P$ and the problem size of the auxiliary model $N_{aux}$. A defining feature of the ES patterns is that the primary model is very computationally intensive and needs petascale or even exascale performance. In terms of the performance model, this means that $T_{pr}(1, N_{pr}) \gg T_{aux}(1, N_{aux})$. Both the primary and auxiliary models can run in parallel, but their scalability can be completely different. For the execution time of the ES scenario we can now write

$$T_{ES} = T_{pr}(P, N_{pr}) + T_{aux}(P, N_{aux}),$$

and for the resulting efficiency of the ES pattern we can write

$$\epsilon_{ES} = \left( \frac{1}{P} \right) \frac{T_{aux}(1)}{T_{aux}(P)} + \frac{T_{pr}(1)}{T_{pr}(P)}$$

$$\approx \left( \frac{1}{P} \right) \frac{T_{pr}(1)}{T_{aux}(P) + T_{pr}(P)}$$

$$= \left( \frac{1}{P} \right) \frac{1}{T_{aux}(P)} + \frac{T_{pr}(P)}{T_{pr}(P)} = \frac{\epsilon_{pr}}{1 + \epsilon_{pr}},$$

with $\epsilon_{pr}$ the efficiency of the primary model, and where for convenience we have dropped the dependence on the problem size. We assume that the primary model scales very well, which seems like a reasonable assumption, given that in the ES case the primary model is the one that requires almost all computing resources and has been optimised sufficiently to run very efficiently on petascale or emerging exascale resources.
We consider two limiting cases. First assume that the auxiliary model also does scale well enough on the available resources, at least so that even on $P$ processors, $T_{pr}(P, N_{pr}) \gg T_{aux}(P, N_{aux})$. In this case we find that $\epsilon_{ES} = \epsilon_{pr}$ and we can simply execute the ES task graph, as drawn in Fig. 2.9(left).

![Figure 2.9](image)

The other extreme would be that the auxiliary model does not scale at all, even to the point that $T_{pr}(P, N_{pr}) \ll T_{aux}(P, N_{aux})$, leading to $ES \rightarrow 0$. This requires a completely different execution of the MCP. Suppose that we could find a pair of processor numbers such that $P = P_1 + P_2$ and $T_{pr}(P_1, N_{pr}) \approx T_{aux}(P_2, N_{aux})$, then we could interleave two instantiations of the ES patterns, combining the execution of a primary model in the first instance with execution of the auxiliary model in the second instance, see Fig. 2.9 (right).

In reality, most cases will be somewhere between these two limiting cases. Here, interesting possibilities appear to optimise both throughput of an ES pattern, not only to have a well-balanced load, but also to optimise energy usage, by allowing adjustment of the clock speeds of the set of processors $P_1$ or $P_2$. In that case $T_{pr}(P_1, N_{pr}) \approx T_{aux}(P_2, N_{aux})$ leads to a whole series of potential solutions of the load-balancing problem, and on that manifold we can then optimise either throughput, and/or energy usage of the execution. The ES MCP algorithm will also take into consideration threshold values, which indicate when to switch from one type of execution to another. It is clear that including a parallel auxiliary model into the picture results in an even richer set of potential solutions, and it is the task of the MCP to find optimal execution scenarios, where ‘optimal’ can be defined in many ways.
2.4 Discussion and Conclusions

We define multiscale computing as the orchestrated execution of coupled single-scale models and scale-bridging methods that together compose a multiscale model. In our experience, corroborated by practice (see e.g., [24, 53]), such componentisation of multiscale computing (as opposed to writing monolithic codes for multiscale models) results in the flexible and efficient development and execution of multiscale simulations. In our opinion, this is the only sensible way to fully exploit all the intricacies of state-of-the-art high end computing resources for multiscale simulations. We have already demonstrated the benefits in distributed computing environments (distributed multiscale computing) [24–26], including clouds [112].

In this chapter we have proposed how multiscale computing can be enhanced to fully exploit current petascale and emerging exascale HPC systems. Whereas the componentisation of multiscale computing has already resulted in a very efficient ‘Lego-based’ philosophy for developing multiscale simulations, as well as the benefit of relative straightforward distributed execution, the challenges of computing on high-end HPC machines require additional functionalities in relation to load balancing, fault tolerance, and energy awareness.

In our vision, on the level of multiscale computing, these exascale challenges can be addressed in a generic way, leading to a separation of concerns. The multiscale application developer can keep relying on the Lego-based approach, keeping in mind that implementations of the single-scale models on HPC resources should already be of very high quality. In many cases, the single-scale models are readily available, as highly efficient public-domain or proprietary codes developed over many years or decades by substantial communities. The execution environment, as offered for instance by the MMSF, should then take care of the load balancing, fault tolerance and energy awareness of the overall multiscale simulation.

The notion of a small number of generic multiscale computing patterns that capture repeating motifs in multiscale computing task graphs facilitates this separation of concerns. Given sufficient information on the performance and energy profiles of the single-scale models, it will be possible to develop MCP software that takes as input the xMML description of the multiscale model and the performance and energy profiles of the single-scale models, and generates execution scripts that a middleware can then use to optimally allocate resources and execute the multiscale simulation. The example that we presented in Section 2.3 sheds some light on how we expect that MCPs can be further developed. A large consortium composed of European Universities, HPC centres and companies are currently implementing these ideas as part of the ComPat project.

Once available, and if successful, the MCPs, in combination with the already existing MMSF, have the potential to dramatically increase the effectiveness with which we can develop, deploy and execute multiscale simulations on emerging exascale resources.
Patterns for High Performance Multiscale Computing

Abstract

We describe our Multiscale Computing Patterns software for High Performance Multiscale Computing. Following a short review of multiscale computing patterns, this chapter introduces the Multiscale Computing Patterns software, which consists of description, optimisation and execution components. First, the Description component translates the task graph, representing a multiscale simulation, to a particular type of multiscale computing pattern. Second, the Optimisation component selects and applies algorithms to find the most suitable mapping between submodels and available HPC resources. Third, the Execution component involves a middleware layer that maps submodels to the number and type of physical resources based on the suggestions emanating from the optimisation part together with infrastructure-specific metrics such as queueing time and resource availability. The main purpose of the Multiscale Computing Patterns software is to leverage the multiscale computing patterns to simplify and automate the execution of complex multiscale simulations on high performance computers, and to provide both application-specific and pattern-specific performance optimisation. We test the performance and the resource usage for three multiscale models, which are expressed in terms of two multiscale computing patterns. In doing so, we demonstrate how the software automates resource selection and load balancing, and delivers performance benefits from both the end-user and the HPC system-level perspectives.

3.1 Introduction

Multiscale modelling and simulation has become a well-established way to study complex phenomena that encompass multiple space and timescales [1]. In this approach, a multiscale model is constructed by combining, or coupling, a collection of single-scale submodels, each of which captures processes on a distinct space and timescale; see e.g., [23, 32, 113]. Multiscale modelling is widely used in most areas of science and engineering [43], such as biomedicine [53, 54, 56],

fusion [24, 49], material science [9, 24], energy [114] and engineering [24, 115]. It is self-evident that any high-fidelity multiscale model must employ substantial high performance computing resources, since the individual single-scale models comprising it have to run on such machines.

In addition to specific multiscale applications, a number of tools and frameworks that assist in multiscale computing have been established. These range from domain-specific frameworks such as AMUSE [116] and OASIS-MCT [117] to solver-specific frameworks such as the MOOSE framework for finite-element codes [118] and fully generic frameworks [1, 3, 27, 28, 32] encompassing related coupling tools such as the Multiscale Coupling Library and Environment 2 (MUSCLE2) [25].

We have previously developed the multiscale modelling and simulation framework (MMSF) [3, 27, 28, 32], which provides a theoretical and methodological framework for constructing multiscale simulations in four main stages. First, we model multiscale phenomena as collections of single-scale submodels, and then decide on which models interact with each other and how. Single-scale models and couplings are presented within a scale separation map, allowing us to describe and compare multiscale models on a conceptual level. Second, we specify the single-scale models, their couplings and interactions using the Multiscale Modelling Language [1, 27]. Third, we convert these definitions to a fully implemented multiscale model, currently relying on MUSCLE2 [25] (although the concepts of the MMSF can also be applied to other coupling environments such as AMUSE [116]). An important property of MUSCLE2 is the separation of concerns that it affords. Submodels do not interconnect with any other components. Moreover, required adaptations are minimal on the level of a submodel in order for it to be incorporated into a multiscale model implemented with MUSCLE2. Fourth, we deploy and execute the multiscale application on a set of computational resources. Developers and users can run different submodels on different machines [23], using for example the QCG middleware [119], a paradigm that we call distributed multiscale computing [23].

Knaps et al. [59] have previously proposed a distributed multiscale computing framework that supports the on-demand execution of micro-scale models coupled to a macro-scale model (very similar to one of the computing patterns we proposed in [63]), using large-scale supercomputing resources. Although their framework has, to our knowledge, not yet been applied outside the domain of materials science for which it was originally created, the authors do propose a general conceptual framework that could be adopted for use in other disciplines. This resonates with our vision of generic multiscale computing environments, where a separation of concerns is achieved between multiscale modelling and simulation on the one hand, and deploying and executing a multiscale simulation in a given HPC environment on the other.

Our "Lego-based" philosophy for the construction and execution of multiscale applications relies on single-scale submodels and their interactions, and results in more degrees of freedom for both programming and executing a multiscale simulation. To efficiently execute multiscale applications on high-end HPC machines, a number of challenges have to be addressed, such as load balance (providing resources to each of the single-scale models), fault tolerance (sometimes instantiations of single-scale models may fail) and energy awareness (depending on properties of single-scale models, potentially also in combination with load-balancing, energy-aware optimisation). Our intention is that these challenges are handled in a generic way, as far as possible avoiding the imposition of that burden on the developers of multiscale applications. Those developers should take care of the scale-bridging mechanisms and the efficiency of the single-scale models, while the challenges of execution within a high performance computing (HPC) environment should be addressed through a generic layer added to MMSF that we call multiscale computing patterns (MCPs) [63].
We defined MCPs as "high-level call sequences that exploit the functional decomposition of multiscale models in terms of single-scale models" [63], and distinguished between three patterns: extreme scaling (ES), heterogeneous multiscale computing (HMC) and replica computing (RC). Each of these patterns is described using a generic task graph that aids in understanding how to best map these patterns to HPC resources. In addition to the generic task graph, an MCP contains performance information about single-scale models, an XML-based specification of the multiscale application named xMML [25], and a set of algorithms and heuristics used to combine this into input files for the execution environment. In this chapter, we report on the design and implementation of the MCP software, and present the first results of executing multiscale simulations using MCPs, including discussions on the added value of using such solutions for high performance multiscale computing. Here, we mainly integrate these MCPs with MMSF to increase the effectiveness by means of which we can develop, deploy and execute multiscale simulations on existing petascale and emerging exascale resources [63].

The MCP software architecture consists of a Description component, an Optimisation component and an Execution component. In the description component, the software uses the task graph of the specific multiscale model, in combination with auxiliary information (e.g., definitions of single-scale models), to identify the type of pattern and create input definitions for the optimisation component. In the optimisation component, the software selects and applies a set of optimisation algorithms to identify a range of efficient mappings of the submodels in the application to specific HPC resources. Lastly, the execution component is a middleware layer that identifies the optimal mapping of submodels to the available resources, taking additionally into account queuing times and resource occupancy. Moreover, the execution component deploys and executes the application, with all of its submodels, on the target resources. Three examples of using Multiscale Computing Patterns software are illustrated and examples of cost functions are worked out, showing that a wide range of variables for multi-objective optimisation algorithms can be chosen. The idea is that Multiscale Computing Patterns software will automatically detect which cost functions and algorithms to select based on the type of pattern and user requirements.

The structure of this chapter is as follows. We describe the MCPs in Section 3.2, and introduce the Multiscale Computing Patterns software and its components in Section 3.3. In Section 3.4, by way of proof of concept, we provide three examples of the use of the Multiscale Computing Patterns software. Finally, we provide a discussion and conclusion in Section 3.5.

3.2 Multiscale Computing Patterns and High Performance Multiscale Computing

In this section, we discuss the concept of multiscale computing patterns and express the MCPs as generic task graphs. For full details, we refer to Alowayyed et al. [63]. Figure 3.1 shows the generic task graphs for all three computing patterns. The extreme scaling pattern represents a type of multiscale model where one primary single-scale model is coupled to a set of serial and/or parallel auxiliary models on any scale as shown in Fig 3.1 (a and b). The primary model\(^1\) is computationally intensive, energy hungry, and highly scalable, whereas the auxiliary models are not. Therefore, the efficiency of this type of multiscale model is highly dependent on the efficiency of the primary model and the primary–auxiliary interactions. Assuming that developers have implemented the primary model efficiently, the main aim is to reach a minimal interference

\(^1\)We assume one primary model here. In practice, ES could consist of more than one primary model.
between primary and auxiliaries. This can be done using load balancing while ensuring minimal communication between primary and auxiliaries. The serial auxiliary model can give rise to strong under-utilisation of available resources (e.g., if it does not scale to a large number of cores), and special mechanisms to handle such situations are required.

The heterogeneous multiscale computing pattern (Fig. 3.1c) represents the typical form of macro–micro coupling, where the numerical solver at the macro-scale level requires input from multiple micro-scale model instantiations (for instance to compute a spatially varying quantity, such as a constitutive equation, say a viscosity in a flow problem). Thus, the number of micro-scale models is dynamic and largely dependent on the dynamic evolution of the macro-scale model. The HMC manager has some control over the number of micro-scale models, by preventing redundant calculations (by storing results of previous micro-scale simulations in an HMC database and where possible extracting results from the database, e.g., by interpolations between results obtained earlier), and spawning extra micro-scale models, when necessary. Typically, the number of micro-scale models will be very large and a single micro-scale run may require substantial computing resources and, hence, dominate computing and energy cost.

In the replica computing pattern a large number of copies of tera- and/or petascale simulations are needed to produce statistically robust results. These replicas are not part of an overarching structure like HMC, but are spawned in the initial step. In this step, the parameter space for parameter sweeping is set. Then, simulations and data processing per replica take place. Both static
and dynamic flavours of RC are considered in Fig. 3.1 (d,e). All replicas execute independently of each other. If a replica (i.e., a simulation) fails, the RC patterns affords some level of fault tolerance, taking into account maintaining the overall statistics. This is the main difference with HMC. On the other hand, HMC and RC are similar in terms of load-balance issues.

3.3 Design of Multiscale Computing Patterns Software

The Multiscale Computing Patterns software consists of three parts: (1) the description part, where the user describes the multiscale application, (2) the optimisation part, where the software predicts and optimises the application performance, (3) and the execution part, where the application is deployed using an underlying resource allocation service (in our case the QCG middleware). We present the components of the Multiscale Computing Patterns software, and their interrelations, in Fig. 3.2.

The logical description and the complete set of requirements of a multiscale application are collected in the Description component. This part relies on concepts from the multiscale modelling and simulation framework. It is helpful to facilitate the work of the end user and provide a single input mechanism for all multiscale applications, as well as detecting the type of MCP. The MMSF xMML description file was extended for replica computing to accommodate the notation of the number of replicas. The Optimisation component determines which MCP optimisation applies, collects required performance figures, and calculates the relevant metrics (e.g., parallel efficiency, throughput, energy usage). Based on these results, a constrained optimisation is performed resulting in a small set of the most suitable execution scenarios, which are passed to the Execution component. The role of the execution component is to select the best allocation plan, based on the availability of the requested resources and cost criteria (time to complete, energy usage), and to start and monitor the execution.

Description component

The Description component (top layer in Fig. 3.2) contains an architecture-agnostic definition of the multiscale application and its main requirements. It builds on concepts from the multiscale modelling and simulation framework. The description component consists of a task graph, submodel definitions, simulation and middleware parameters and user information, all feeding into the Translation service.

The task graph is expressed in the form of a highly adaptable textual description (xMML, see [25]) that is used to detect motifs (repetitive submodels and dependencies). The task graph is also needed to observe workflow-related issues such as the expected frequency of communication between submodels.

Submodel definitions contain all the information required for a single-scale model to run. This includes information on submodel-specific dependencies, and the resource requirements for each submodel (e.g., mandatory use of GPU architectures, or a minimal memory requirement per core). The Description component may rely on previously developed tools such as MAD/MaMe [120], and can already leverage existing configuration information from the FabSim automation environment [102].

All the simulation and middleware parameters are collected in a separate component. This includes input parameters, the required environment modules and resource limits for the overall
CHAPTER 3. PATTERNS FOR HIGH PERFORMANCE MULTISCALE COMPUTING

Figure 3.2: Architecture of the Multiscale Computing Patterns software. The dashed-line blue boxes represent external components (which either exist separately or are under development).

simulation (all submodels and the coupling library). Also, this component holds all information needed to compose the multiscale simulation (e.g., using MUSCLE2) and to execute the simulation (e.g., using QCG middleware [121]) using (distributed) HPC infrastructures such as the Experiment Execution Environment (EEE). This component is designed such that existing known machine configurations from FabSim (machines.yml) can be directly reused in the context. Similarly, user-specific information can be directly reused from existing FabSim configurations (machines_user.yml). We present an example of the reuse of FabSim information within this context as part of the Binding Affinity application described in Section 3.4.

These three pieces of information are then supplied to the Translation service, which merges and converts them into a format suitable for the Optimisation component. Currently, the translation tool is application specific, and produces two xml files as output. One file, matrix.xml, is shown in Listing A.1 in A and contains templated information from the submodel definitions. The other file, multiscale.xml shown in Listing A.2 (A), has information from the simulation and middleware parameters.

Optimisation component

The main software tool within the Optimisation component is the Pattern-Driven Planner. This tool requires input from both the Translation service as well as the node description list. The node description list is updated regularly to reflect the current status of available nodes in the targeted supercomputers, and contains information of node types. A single node type represents
design of multiscale computing patterns software

a set of nearly identical nodes in terms of hardware configuration (e.g., processor type). The
node types should be defined based on knowledge gathered a priori by the middleware from the
infrastructure provider. Table 3.1 shows an example of node types.

Table 3.1: Example of node types, an input to the Pattern-Driven Planner. Note that RAM per node is in
gigabytes. Processor type 1 Intel(R) Xeon(R) CPU E5-2697 v3@2.60 GHz, 2 Intel(R) Xeon(R) CPU E5-2680
@ 2.70 GHz, 3 Intel(R) Xeon(R) CPU E7-4870 @ 2.40 GHz and 4 Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60
GHz

<table>
<thead>
<tr>
<th>Type name</th>
<th># of nodes</th>
<th>Processors per node</th>
<th>Cores per node</th>
<th>RAM per node</th>
<th>Processor type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eagle</td>
<td>haswell_64</td>
<td>492</td>
<td>64</td>
<td></td>
<td>(1)</td>
</tr>
<tr>
<td></td>
<td>haswell_128</td>
<td>460</td>
<td>2</td>
<td>28</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>haswell_256</td>
<td>52</td>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SuperMUC</td>
<td>thin</td>
<td>9216</td>
<td>2</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>fat</td>
<td>205</td>
<td>4</td>
<td>40</td>
<td>256</td>
</tr>
<tr>
<td>Stfc</td>
<td>default</td>
<td>118</td>
<td>2</td>
<td>16</td>
<td>64</td>
</tr>
</tbody>
</table>

The second layer contains the Pattern-Driven Planner and the Performance Estimator components. The Pattern-Driven Planner component collects measurements and/or predictions of performance of submodels, under various execution scenarios, from the Performance Estimator. Then, it uses this information to compute required cost functions (e.g., efficiency, throughput, energy usage, or a combination) on available resources. Given the specific MCP and all other available information, the Pattern-Driven Planner performs a constrained optimisation against these cost functions, and provides a selection of particularly suitable execution scenarios to the Execution component. The Execution component will then select the optimal execution plan based on chosen specified cost criteria (time-to-completion, energy consumption), by taking into account additional information only available to the middleware (e.g., estimated queuing time, live information on availability of resources, etc.).

The Measurements and Architecture Modelling components respectively store and calculate submodel performance information as a function of the chosen number of cores/nodes and problem size. The Single Scale Performance Model captures the scalability of submodels with respect to problem sizes and number of processors. The Performance Estimator, in turn, relies among others on the Single Scale Performance Model to obtain, interpolate and/or calculate performance results for the multiscale model. For example, this could be achieved by interpolating between performance results of adjacent problem sizes in the multiscale model and/or, relying on performance models, to predict the performance using a core count for which no measured values have yet been obtained.

In the Measurements component, the overall cost of the submodel as a function of problem size is measured for 1 to n cores on the first node in a specific node type, and for 2 to N nodes assuming full occupancy on each node for a given number of iterations. In the baseline case, the cost is represented as execution time, but note that these calculations can also be done for other metrics of cost such as energy. The actual measurements can be obtained from tools designed specifically for performance profiling tools, such as Allinea MAP [122], the tool of choice in our research. A template of measurements is shown in Listing A.3 in A; this measurement listing.
might contain specially marked values (i.e., NA), for node types where a specific single-scale model is not supported.

The Architecture Modelling software is established to provide predictions for existing machines, but also for non-existing emerging exascale configurations. This allows users to assess how MCPs could optimally benefit from such hypothetical machines, or contribute in the co-evolution of such new architectures.

Based on performance results from the Performance Estimator, the Pattern-Driven Planner groups types of nodes into classes depending on the similarity of performance figures, type of computing pattern and cost criteria (e.g., efficiency, makespan time, energy usage, resource usage, etc) computed as cost function. Then, using multi-objective optimisation, the tool will generate a small number of alternative execution scenarios. The importance of the alternatives here is to give the Execution component the freedom to choose from a set of resources with comparable performance per submodel, depending on the availability of these resources as well as on the variation in queue times. We will enhance this component to extend the patterns with capabilities to also consider issues related to energy awareness and fault tolerance. All in all, for each pattern we will formulate constrained optimisation problems that as output will deliver alternative execution profiles to the Execution component.

The exact output of the Pattern-Driven Planner to the Execution component will be several allocation plans and other requirements, as described in the next section, to run multiscale applications. Here, the output file holds information about the kernels and corresponding helpers, the classes of node types and a set of allocation plans. An allocation plan is a specific mapping of the multiscale model to resources. Listing A.4 in A shows the template of the node classes and the parts of the allocation plans.

**Resource allocation and Execution component**

The responsibility for the Execution component is twofold. First, it needs to select the best allocation plan from the plans provided by the Optimisation component. The selection pertains to the mapping of computational kernels to a specific set of physical resources of defined types, taking into account the (sometimes conflicting) requirements of users and resource providers. Second, once an optimal plan has been selected, this component needs to ensure the efficient and reliable execution of the application within the distributed heterogeneous infrastructure.

The Execution component is mainly provisioned using the QCG environment\(^2\) [120], a mature middleware system deployed in many HPC centres across Europe. QCG delivers a set of ready-to-use components that can be installed and managed at each site, irrespective of the internal policies or local queuing systems. To fulfil the expectations and objectives of both users and resource providers, QCG features an extendable brokering service that allows for customised brokering algorithms and strategies. In addition, QCG provides support for advance reservation, co-allocation and workflows, enabling the execution management of multi-kernel applications, with both cyclic and acyclic dependencies, on a distributed e-infrastructure [25, 119].

Deploying multiscale simulations on production e-infrastructures gives rise to a number of challenges that are difficult to anticipate prior to the Execution component coming into effect. For example, the user objective for an immediate job start, e.g., through means of advance reservation, needs to be harmonised with the provider’s objective for high resource utilisation. In addition, the Pattern-Driven Planner provides plans that are likely to be optimal from a user perspective,\(^3\)www.qoscosgrid.org
but have not yet incorporated the constraints imposed by the presence of other workloads in the e-infrastructure environment.

The QCG Pattern-aware Scheduler (which is part of QCG Broker) calculates which of the plans provided by the Pattern-Driven Planner is optimal with respect to the objectives of all involved stakeholders. In doing so it takes into account the current and historical load on e-infrastructure resources, including both the occupancy of the actual resources and the queue lengths. The Pattern-aware Scheduler can perform this optimisation with respect to required cost criteria, either a single time-to-completion criterion or a combination of two criteria, total energy expenditure and time-to-completion. Here, the time-to-completion is calculated by adding the predicted queuing time (predicted by Queue Time Prediction Service to QCG) and execution time (provided by the optimisation component). In the energy optimisation case, QCG Scheduler firstly selects a set of candidate plans that finish according to the QCG time-to-completion prediction in the requested period of time and then it selects an optimal plan with the minimal total energy expenditure (calculated and given by the Optimisation component). Through its direct integration with the QCG environment, the Pattern-aware Scheduler accounts for the multi-kernel nature of multiscale applications and the fact that each kernel may behave differently in the context of performance and energy usage when executed on different resource types [109, 121, 123].

The QCG Pattern-aware Scheduler relies on a plugin-like architecture to gather all required information (see dashed boxes in the Execution component of Fig. 3.2). For example, the scheduler uses the Queue time metrics plug-in to get precise knowledge about the expected queue time on available resources. This plug-in is integrated with external resource-level components, in this case the Queue-time Estimation service. Similarly, we are planning to implement an Energy metrics plug-in and combine it with the QCG Pattern-aware Scheduler.

As the new brokering module uses new types of input parameters to specify the requirements of the MCPs, we have extended the job description interface and revised several internal schemas used to exchange information between the components in the QCG Broker service. We present several key fragments of the extended description in A.5. Here, all jobs described using a patternTopology element will be processed using the new scheduling engine.

Based on the result of the QCG Pattern-aware Scheduler, the QCG Job Controller module prepares the execution environment by transferring input data and starting the job submission to one or more distributed resources. The resources in our e-infrastructure are made accessible to the QCG Job Controller using services implementing the Basic Execution Service (BES) interface [124].

QCG Job Controller contains a set of specific adaptations to address the requirements for efficiently executing high performance multiscale simulations using high-end e-infrastructures. Both the QCG Broker interface and the core capabilities of the service components have been extended to support a range of pattern-based multiscale jobs. Specifically, to allow efficient execution of the replica computing pattern applications, we have incorporated two additional QCG mechanisms: workflows and job arrays. We incorporated a modified version of existing workflow mechanisms [125], eliminating the need to transfer data between subsequent tasks executed on the same resource, and simplifying the execution of workflows in parameter sweep tasks. The job array's functionality allows a set of independent tasks to be run on a resource and be considered as a single QCG task. In the replica computing pattern these sets of subtasks can be scheduled by the middleware to be executed on various clusters, thereby balancing the overall load on the infrastructure. Job arrays not only help to reduce the management complexity of all
tasks executed separately, but increase the overall throughput of the system and decrease the total time-to-completion of a simulation.

### 3.4 Applications of the Multiscale Computing Patterns Software

In this section, we present three exemplar applications from different scientific domains (one from fusion research and two from biomedicine, being cell-based blood-flow modelling and the Binding Affinity Calculator BAC) to demonstrate the capabilities and practical usage of the MCP software, and the benefits in terms of application performance. Our applications are mapped to two different computing patterns, with fusion and cell-based blood-flow modelling mapped to the extreme scaling pattern and BAC to the replica computing pattern. Application for HMC is presented in the next chapter. In addition, details of the required steps and various code snippets at each level of the software stack are presented from the perspective of the application developer. All performance figures presented are measured on two supercomputers that participated in the studies, namely SuperMUC\(^3\) (Tier-0 HPC from Leibniz-Rechenzentrum, Germany), and Eagle\(^4\) (Polish national grid clusters from Poznan Supercomputing and Network Center, Poland). Further details are listed in Table 3.2.

| Table 3.2: Resources used for the measurements in Sections 3.4 and 3.4. |
|-----------------------------|-------------------------|------------------|
| Resource | CPU architectures | Cores |
| SuperMUC | Intel(R) Xeon(R) CPU E5-2680 | 147456 |
| | Intel(R) Xeon(R) CPU E7-4870 | 8200 |
| Eagle | Intel(R) Xeon(R) CPU E5-2697 v3 | 2408 |

**Extreme scaling**

In ES, the ultimate goal is to ensure minimal interference between the primary model and the auxiliaries. It may happen (as in the example of the cell-based blood-flow simulation) that the auxiliary models induce large waiting times for the primary model, thus potentially wasting resources and reducing resource usage. The Multiscale Computing Patterns software detects this situation automatically, and then interleaves two multiscale simulations, executing both at the same time [63]. This mechanism would increase the resource usage efficiency. For ES, the efficiency of the multiscale model \( \epsilon_M \) can be calculated as [63]:

\[
\epsilon_M = \frac{\epsilon_p}{T_{aux}(P)} + 1,
\]

and the resource usage efficiency (R) as:

\[
R = \frac{\sum_i T_i P_i}{T \sum_i P_i}.
\]

---

\(^3\)www.lrz.de/services/compute/supermuc

Where $P_i$ is the number of cores used for submodel $i$, $T_i$ is the execution time on submodel $i$ excluding waiting times, $T$ is the total execution time including waiting times, and $\epsilon_p$ is the efficiency of the primary model.

**Fusion application**

Nuclear fusion has the potential to produce clean and carbon-free energy, as physicists hope to demonstrate with ITER, which is a tokamak device that uses magnetic fields to confine plasma. However, the grand challenge of long-term plasma confinement requires the understanding of interactions between very small-scale turbulence and large-scale plasma behaviour [22, 49]. Therefore, having a robust multiscale computing scheme to study this interaction has become a vital goal in the fusion community. Our targeted Fusion application simulates the time evolution of a plasma’s 1-D profiles (for instance, electron temperature) in the tokamak core with a transport code, while under the influence of anomalous transport coefficients computed by a 3-D turbulence code and periodic 2-D equilibrium reconstruction [22]. The transport, turbulence, and equilibrium codes are submodels developed separately and are well benchmarked. These submodels share a common data interface and are embedded into MUSCLE2 as kernels, which allows for straightforward coupling through a simple and configurable script as described in [49]. Such simulation is essentially multiscale in time, and corresponds to the ES computing pattern depicted in Fig. 3.1 (b). The turbulence code is the primary submodel in this application because it requires the vast majority of the computational power compared with the other submodels.

The starting point in the Description component of the software is to compose a task graph in xMML format. This text file contains the list of submodels involved, time and space scales and input/output data of each submodel, and coupling between submodel pairs through their respective input/output data. If desired, the user can deploy the jMML tool [25] to generate the task graph from the xMML [113]. Besides visual representations, the jMML tool can turn the content from a task graph into a skeleton configuration for MUSCLE2. The designer of the coupled application can implement submodels as MUSCLE2 kernels and other simulation parameters (either global or specific to a kernel) into the MUSCLE2 configuration file [25]. Note that at this stage, the user can directly connect to a cluster where all executables, libraries and input data are present, and write an ad-hoc script to be submitted to the local batch queue system. However, the burden of manually adjusting the configuration and selecting the optimal cluster lies with the user every time they want to run a simulation. The MCP software has features that relieve these burdens from the user by automatically selecting the best configuration for a given performance metric, as described in further detail in the remainder of this subsection.

The task graph is submitted and parsed by the Translation service along with other specifics provided by the developer, such as additional submodel definitions, details on middleware, simulation parameters, and user information. In the current implementation, the Translation service is a python script which, as a result, creates two template xml files: matrix.xml and multiscale.xml. Matrix.xml contains information related to single-scale submodels, such as measurements of their performance. An example of benchmark data on scaling of the primary submodel for two types of nodes is shown in Fig. 3.3. Multiscale.xml contains information related to the coupled application. These two templates are pre-filled with information from the xMML file and can be completed by the application designer.

Next, the outputs of the Translation service (matrix.xml and multiscale.xml) are passed on to the Pattern-Driven Planner, which in turn generates an XML job script for the selected
middleware (the QCG). Currently, the Pattern-Driven Planner proposes three optimal plans that minimise the cost, and an example of such plans is shown in A.5. Currently, these plans are drafted based on the measurements of runs performed manually by the application designer. The next stage will be to enhance the Performance Estimator such that it can benchmark on-the-fly and interpolate on settings for which no performance data is available.

Table 3.3: Performance for ES applications, namely Fusion and cell-based blood-flow modelling (RBC). T is the execution time (excluding waiting times) for primary (Pr) and auxiliaries (aux) on \( P_{Pr} \) and \( P_{aux} \) number of cores in seconds, \( \epsilon \) is the efficiency for the primary (Pr) and the multiscale model (M) and R is the resource usage.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Host</th>
<th>( P_{Pr} )</th>
<th>( T_{Pr} ) (S)</th>
<th>( P_{aux} )</th>
<th>( T_{aux} ) (S)</th>
<th>( \epsilon_{Pr} )</th>
<th>( \epsilon_{M} )</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fusion</td>
<td>SuperMUC</td>
<td>1024</td>
<td>49017</td>
<td>1</td>
<td>780</td>
<td>1.0618</td>
<td>0.9774</td>
<td>0.91</td>
</tr>
<tr>
<td>RBC</td>
<td>Eagle</td>
<td>1036</td>
<td>1936.7</td>
<td>168</td>
<td>1531.6</td>
<td>0.7066</td>
<td>0.3946</td>
<td>0.367</td>
</tr>
<tr>
<td>RBC_alt</td>
<td>Eagle</td>
<td>1036</td>
<td>3081.34</td>
<td>168</td>
<td>3081.34</td>
<td>0.7066</td>
<td>0.3954</td>
<td>0.746</td>
</tr>
</tbody>
</table>

Finally, the job script from the Pattern-Driven Planner is submitted to the QCG. QCG selects one of three plans and starts the simulation on the system(s) involved. For the Fusion application, and in general for most ES applications, it is more sensible to select a plan in which all submodels run on a single site, for auxiliaries do not require many resources. In that case, we should only care about serialisation due to serial auxiliary models and how that could impact the execution [63]. Also note that the speed-up of the primary model is super-linear because the efficiency was calculated with 64 cores instead of one core, which may lead to latency hiding. For Fusion, the time of the primary model spent in waiting for the auxiliaries is not very large, as shown in Table 3.3, so no additional actions are required, and, therefore, the resource usage is high.

In particular, QCG selects the thin nodes in SuperMUC to run the fusion simulation (see Table 3.3). A production run with 1000 iterations using 2048 cores was completed successfully using the software scheme described earlier. The entire run was completed in approximately 11.1 hours, or 22733 core hours. Among the three submodels, the primary submodel (a turbulence
code based on gyrofluid theory) took about 17 seconds per iteration, while the transport and equilibrium auxiliary submodels took 1 and 3 seconds, respectively. However, the fusion plasma in this particular example needs approximately 4000 iterations to reach a state of equilibrium. Therefore, improving efficiency becomes essential as future simulations require more computing time. The current simulation couples the submodels in series. One way to speed-up the simulation is to run auxiliary submodels in parallel when possible, which is theoretically the case for the timescale-less equilibrium submodel. This idea is preliminary and its validation is necessary before such transformation is added as a possible optimisation technique in the Pattern-Driven Planner.

The ultimate goal for the Fusion application is to use a more sophisticated turbulence model, namely replacing the gyrofluid model with a gyrokinetic model, to simulate plasma in the core of a tokamak. In addition, the ability to simulate plasma of a much larger volume would be necessary to understand possible instabilities that could destroy plasma confinement in the ITER tokamak. These goals require an extensive amount of computing resources, as well as intelligent and highly optimised coupling approaches. The Multiscale Computing Patterns software has demonstrated initial success with a smaller-scale problem. With further improvements, we envision that these patterns can efficiently handle future exascale calculations involving ITER and gyrokinetic simulations.

Cell-based blood-flow simulation

In this application, we couple continuous blood-flow simulations implemented in Palabos\(^5\) (a fully parallelised open-source lattice Boltzmann model) to cell-based blood-flow simulations implemented in the HemoCell\(^6\) suspension simulation framework (an Immersed Boundary–lattice Boltzmann model (IB-LBM)) \([91, 98, 126]\). Specifically, we couple two continuous fluid fields \((C_L \text{ and } C_R, \text{ which are serial auxiliary models})\) to the inlet and outlet of HemoCell, in order to provide the correct in- and outflow conditions to the more expensive suspension simulation \((P, \text{ the primary model in this application})\), and to keep the domain for the cell-based blood-flow simulation as small as possible. This application has also been coupled using MUSCLE2. The performance measurements are shown in Fig. 3.4. As is clear, in this case the auxiliary models \((C_L \text{ and } C_R)\) require a small amount of computing and only execute on a small core count. However, the primary model, HemoCell, is compute hungry, but at the same timescales very well to a much larger core count, even to the point that the execution time of the primary becomes comparable to the execution time of the auxiliary submodels. This situation was analysed in \([63]\) and calls for a more advanced scheduling of the pattern, basically interleaving two instantiations in order to make best use of the available computing resources. The MCP software is able to orchestrate such more advanced scheduling of multiscale applications.

Table 3.3 shows that the resource usage for running this application is 0.35. This is due to the large waiting times of both primary and serial auxiliaries in the naive scheduling, which means wasting 1122 core hours (1.08 hours per core) for primary and 1755 (12.5 hours per core) for auxiliary models by doing nothing but waiting. To solve this, we interleave two different instantiations with each other, as proposed in \([63]\). This mechanism was coordinated using wait/notify semantics \([24]\). By doing so, we doubled the resource usage efficiency by reducing the wasted cores to 887 and 152 core hours for primary and auxiliary models, respectively. By

\(^5\)www.palabos.org

\(^6\)https://www.hemocell.eu/
implementing more advanced load-balancing algorithms and selecting the right number of cores for the primary and auxiliary models, we can increase the resource usage efficiency even more. We are currently realising such more advanced features of the MCP software.

**Replica computing (Binding Affinity Calculator)**

The procedure for replica computing is similar to that for extreme scaling (Section 3.4). The starting point for all RC pattern applications is the task graph, via an xMML textual description. A “multiplicity” tag in the “instance” node of the xMML description indicates that multiple instances (replicas) are required for that submodel. The Translation service detects the presence of this tag, identifies that the RC pattern is required and that the associated cost function in the Multiscale Computing Patterns software should be invoked.

The Translation service uses submodel definitions in separate files. To illustrate this, we describe the process for the Binding Affinity Calculator (BAC) [100], an automated molecular simulation-based free energy calculation workflow tool, which we use to calculate ligand–protein binding affinities. Rapid and accurate calculation of binding free energies is of major concern in drug discovery and personalised medicine. The underlying computational method is based on classical molecular dynamics (MD). These MD simulations are coupled to the molecular mechanics Poisson–Boltzmann surface area (MMPBSA) method to calculate the binding free energies. For purposes of reliability, ensembles of replica MD calculations are performed for each method, and we have found that about 25 of these are required per MD simulation in order to guarantee reproducibility of predictions. This is due to the intrinsic sensitivity of MD to the initial conditions, since the dynamics are chaotic. Therefore, BAC is an ideal example of the replica computing pattern. BAC consists of a workflow where, within each replica, the output from one submodel (NAMD) is used as input to the next submodel (AmberTools). For more information, we refer to [100, 127].

BAC previously used the FabSim [102] tool extensively to perform simulation runs and, therefore, we have added an option to the Translation service to allow the matrix.xml and multiscale.xml files to be completed (as much as possible) through reading of FabSim configuration.
files. This demonstrates the potential versatility of our MCP approach, which should enable relatively straightforward integration with the existing multiscale execution environment, in this case, FabSim. For example, it uses the machines.yml configuration file from FabSim, which lists the configuration settings of submodels on remote resources (e.g., location of libraries and required execution flags). Additional information specific to the Translation service (and not required by FabSim) can also be added to this file, including restrictions on the submodel (GPU / CPU compatibility, max / min number of cores, etc.). This allows submodel information to be reused if it is required for different multiscale applications. Then, FabSim-compatible YaML files are used to assist the completion of matrix.xml and multiscale.xml. BAC currently does not use a coupling library (such as MUSCLE), so no additional files are required. However, in the future we foresee hybrid MCPs, where each replica could for instance be a full-blown ES by itself, and then such additional information would be needed.

Following the procedure outlined for the ES pattern, the user passes matrix.xml and multiscale.xml to the Optimisation component. Unlike the ES pattern, the user does not need to specify the required number of cores for the overall simulation. This is decided by the Performance Estimator by calculating the cost function.

The method used to find a cost function for RC that will generate resource allocation plans is different to that for ES. First, there is an obvious trade-off between the number of replicas that must be executed, the minimum number of cores that one single replica needs, and the total number of cores available for the overall job. The performance data for RC uses the minimum time per replica for different node types in different hosts, as shown in Fig. 3.5 for a single BAC replica. This data is collected in the Single Scale Performance Model. In the simplest cost function, where we consider only time to solution, all replicas would be run concurrently on the node with the shortest running time per replica. However, there are several constraints that the Performance Estimator must also consider such as queue constraints (number of concurrent jobs, time limitations node availability and queueing time).

Most supercomputers have a limit on the number of jobs that can be run or queued at any moment in time per user. For example, on the SuperMUC machine, the maximum number of jobs that can be run concurrently on the thin nodes in the "general" queue is 8, while there are no restrictions on the Eagle machine.

As an example, if we have two RC applications that require 40 and 80 replicas respectively, the Pattern-Driven Planner needs to calculate which is faster: running all replicas at one supercomputer, that is, SuperMUC (while taking into account the constraint of concurrently running 8 jobs per user) or distributing the jobs among different hosts, for example, across SuperMUC and Eagle, using the functionality in QCG to run across multiple resources. To illustrate how this could be coordinated, let us take the hypothetical situation that there is also a 12-job limit on concurrent jobs running on Eagle to mimic the workload. In Figure 3.6, we show the time-to-completion as a function of the number of "batches" running on SuperMUC, where a "batch" is defined as a set of 8 concurrent jobs running on SuperMUC. The remainder of the replicas are run on Eagle (again in "batches" of up to 12 jobs).

Figure 3.6 shows our estimate, that for 40 replicas, the shortest time-to-completion is for 2 "batches" to be run on SuperMUC, while for 80 replicas, the minimum time-to-completion is for 4 "batches" to be run on SuperMUC. This assumes that all replicas take the same time (the shortest time-to-completion from our benchmarking), that all the replicas are independent (no communication) and that each "batch" runs directly after the other. It is clear there is a limitation to this model; it will only be realistic if the time spent in the queue is very short. Otherwise, the
time-to-completion could be very different to that predicted in Fig. 3.6, and we could envisage the most efficient split in replicas across resources being completely changed if the queuing times are very different across the resources. The estimation of queueing time will be investigated and then incorporated into the middleware in the future (as described in Section 3.3).

The output file description to the Execution component is unified among all computing patterns as described in Section 3.3. QCG also has the ability to distribute replicas to the intended machines and gather the results in one place. Figure 3.7 shows timings of test BAC runs. In these studies, we run 10 replicas across two supercomputers, SuperMUC and Eagle. By running 8 replicas on SuperMUC and the rest on Eagle, we reach the least time to completion.

To quantify this speed-up, we would compare the best timing of distributing replicas $T_{dist}$, with the best timing of running them on SuperMUC (with batch time) $T_{local}$. The speed-up
is calculated as:

\[ S = \frac{T_{\text{local}}}{T_{\text{distr}}} \]

and the speed-up is 1.2 for our set of studies. This means that at the moment of running this set of replicas, we would gain a speed-up due to the varied queuing time. The queuing time will be predicted and hosts will be automatically selected by QCG based on the knowledge of runtime and queuing time as stated before. Table 3.4 summarises the results from the BAC application for time-to-completion runs in Fig. 3.7.

Table 3.4: Performance model for RC application, namely BAC. N is the total number of replicas, \( P_R \) is the number of cores per replica, \( T_R \) is the time per replica in seconds, \( T_{\text{local}} \) and \( T_{\text{distr}} \) are the shortest total simulation time (including queuing times) for local and distributed runs and \( S \) is the speed-up.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>N</th>
<th>( P_R )</th>
<th>( T_R ) (S)</th>
<th>( T_{\text{local}} ) (S)</th>
<th>( T_{\text{distr}} ) (S)</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAC</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>29</td>
<td>24</td>
<td>1.208</td>
</tr>
</tbody>
</table>

3.5 Discussion and Conclusions

We have introduced and described the Multiscale Computing Patterns software, which extends the Multiscale Modelling and Simulation Framework to enable high performance multiscale computing based on three generic patterns. We demonstrated its usage and added value for three different types of multiscale applications: fusion and cell-based blood-flow simulation, both as examples of extreme scaling, and binding affinity calculation as an example for replica computing. In addition, these multiscale models are based on different coupling approaches, including MUSCLE2, as well as coupling using scripts and the FabSim automation toolkit.

We implemented and demonstrated the extreme scaling and replica computing computing patterns. The third computing pattern, heterogeneous multiscale computing, will be implemented, discussed and demonstrated in the next chapter. In the current implementation, each of the demonstrated applications highlights specific strengths of our software approach. For the Fusion...
application, the software abstracts the complication of HPC and chooses the most appropriate number of cores to obtain the required cost criterion (i.e., time-to-completion). For blood flow, our approach enabled the use of double the resources otherwise accessible. Lastly, for binding affinity calculations, our approach serves to abstract away the choice as to whether the replicas should all run on one and the same computer or be distributed across multiple computers. This automated scheduling approach, which recommends execution across two resources, delivers a time-to-completion speed-up of 1.2 compared with the scheduling of all replicas on a single resource.

The Multiscale Computing Patterns software maintains a separation of concerns in three areas. The top layer, the Description component, represents the logical description of the multiscale model. This is the part that is most “visible” to the application users and developers. The Optimisation component is focused on performance aspects, and provides a number of optimisation criteria based on the type of the multiscale computing pattern and the required criteria. Finally, the Execution component integrates a range of functionalities from the underlying e-infrastructure, and uses the information from the Description and Optimisation components to create and run execution scenarios, each optimised either for minimal time to completion, or minimal energy consumption (given a fixed time-to-completion requirement). This modular implementation helps multiscale model developers to concentrate on optimising the single-scale models of which the application is comprised, without needing to go into details about the HPC machines. The developer can choose the optimisation criteria required.

In this work, we have assembled a range of powerful functionalities for optimising and deploying multiscale applications on large-scale HPC infrastructures operating at the multi-petascale, and presented an application-agnostic approach that reduces the development effort required for these purposes. We plan to release the software described here shortly. Generic approaches to high performance multiscale computing are highly sought after across scientific disciplines, and indeed we have already begun propagating the first elements of our approach to other application domains such as astrophysics and materials modelling.
4.1 Introduction

After defining, discussing, and providing proofs of concept for two computing patterns (namely extreme scaling and replica computing) in the previous chapter, we will now move towards the third and most dynamic computing pattern, heterogeneous multiscale computing (HMC) [63]. The HMC pattern implements a family of multiscale models which, using the MMSF terminology (see Chapter 2 and [24, 63]), are single domain with multiple and dynamic instantiations of the micro-scale dynamics, utilising a call/release coupling template [24]. The heterogeneous multiscale method [99, 129] represents the most obvious multiscale model that fits the HMC patterns, but other examples would include running uncertainty quantification on extreme scaling applications using so-called semi-intrusive algorithms [130].

In this set of multiscale applications, a complex phenomenon is modelled by employing a numerical solver for the macro-scale equations and obtaining missing properties (e.g., constitutive equations) from suitable micro-scale simulations. Hence, the macro-scale model(s) are coupled, usually for each iteration, with a large and dynamic number of micro-scale models (Fig. 4.1).

The HMC pattern is based on, and inspired by, the hierarchical multiscale method (HMM) [99]. The primary potential and advantage of HMM is capturing of the dynamics at the macro-scale level by considering some microscopic details of the problem. Thus, HMM is a modelling technique used to numerically solve multiscale problems by coupling multiple submodels together, each of which solves a component separately. An overall macro-scale model then emerges when the separate submodels are combined. Micro-scale models are employed to resolve each unknown component of the problem separately and return the result to the macro-scale model. In this case, heterogeneous suggests that the problem is multi-physical in nature [131]. Frameworks taking into account the computational aspects of HMM, certainly in relation to HPC, are rare [59, 63]. For this reason we propose heterogeneous multiscale computing.

For illustration purposes, a good example of HMM is provided by a flowing suspension [31]. A lattice Boltzmann fluid model of local velocity $u$ and an advection–diffusion model of
local particle density $H$, representing the macro-scale models, are coupled with a set of fully resolved 3-D lattice Boltzmann suspension model(s). These micro-scale models compute the local viscosity $\nu$ and the diffusivity tensor $D$ from a complete run on each, or many lattice point(s) in the macro-scale model to be used in the next time step [31]. This example will be used in this chapter as proof of concept and is shown in Fig. 4.2.

From a computational point of view, the potentially large number and dynamic nature of the number of required micro-scale models, which are determined at runtime, can become a bottleneck in production runs [63]. Moreover, micro-scale models can be computationally
intensive, such as in the 3-D example considered earlier. The number of micro-scale models also depends on the spatial properties of the macro-scale model [31].

A common solution to reducing the number of micro-scale simulations is to utilise a clever database, or surrogate model, as an intermediary layer between the scales to prevent re-computing parameters that are already known, and to use the already computed quantities at the micro-scale to build a surrogate model. The database, in conjunction with an HMC manager, restores previously computed data, interpolates them where necessary using the surrogate model, and provides input to the macro-scale model(s). Depending on the state of the surrogate model and the history of the data required, this mechanism can reduce the number of micro-scale simulations significantly.

Generally, this solution is practical and has been used in multiple fields [132, 133]. The exchange of data should not be a bottleneck in this solution, because the exchange between scales usually involves a few floating-point numbers that represent specific properties. However, the dynamic number of micro-scale models should be mapped efficiently to available hardware resources to avoid potential load-balancing issues [63].

The HMC manager manages the surrogate model. In this process, the macro-scale model sends a request to the HMC manager for the required quantities. The manager then consults the surrogate model for the required information. For this purpose, a user-defined script is implemented to decide whether the cached or interpolated data is sufficient. If the available data is not sufficient, the manager will start new micro-scale models to obtain more accurate results. To prevent the manager itself from becoming the bottleneck, we will assume asynchronous I/O and separate computing resources for the HMC manager.

From a computing perspective, the main difference between the multiscale applications in HMC and those in replica computing is the usage of the dynamic surrogate model and the dynamically changing need for computational resources for the micro-scale simulations. However, although the main benefit of the dynamic nature of the surrogate model is to decrease the computational cost, it does potentially increase load imbalance because the number of single-scale models, at each macro-scale iteration, is dynamic [133]. The purpose of the HMC pattern is to mitigate these issues and provide a level of control over the execution that aims to minimise load imbalance and maximise resource utilisation over the complete HMC run. Our proposed solution is to use the resources dynamically supported by the architecture, with the assistance of tailored scheduling controlled by the pattern. For the architecture, we will use the pilot job mechanism. The pilot job, RADICAL-Pilot, for example [134], is a normal job submitted to supercomputers to reserve resources and use the resources as a unified unit. The QCG Pilot Job Manager system has a similar mechanism by allowing the management of the resources on the application level. The pilot job is akin to a traditional job array, but it allows the scheduling and execution of small and dynamic jobs with different resource requirements. In the next section, we will focus on load balancing in micro-scale models in HMC.

4.2 Load Balancing in HMC

In this type of multiscale applications, we will utilise a pilot job running on $P_{\text{total}}$ processors for a time $T$, where $P_{\text{total}}$ and $T$ are decided by the user at compile time. Let the macro-scale model ($M$) execute on $P_M$ processors, the surrogate model ($DB$) execute on $P_D$ processors, the HMC manager ($HMC$) on $P_H$ processor and $\eta$ micro-scale models ($\mu$) execute on $P_\mu$ processors. For
the macro-scale model, the HMC manager and the surrogate model, the processor allocation is static, while for the micro-scale models it is dynamic.

Ultimately, the number of micro-scale models \( \eta \) to be executed is dynamic and changes at every macro-scale time step, depending on how effective the surrogate model is in predicting the missing quantities. Thus, \( \eta(t) \) is the number of micro-scale models at time \( t \). Its range is \((0 \leq \eta \leq \text{DoF})\), where \( \text{DoF} \) is the total number of degrees of freedom on the macro-scale model that are coupled to micro-scale simulations. This could, for example, be the number of lattice points in a flow simulation where in each lattice point the viscosity is unknown and needs to be obtained from a micro-scale simulation.

The number of successful calls to the database per time step, which replaces the need to generate micro-scale jobs, is \( \eta_D(t) \). This will reduce the number of micro-scale models to

\[
\eta(t) = \text{DoF} - \eta_D(t).
\]

We define \( g(t) \) as the performance of the surrogate model,

\[
g(t) = \frac{\eta_D(t)}{\text{DoF}},
\]

and then write the actual number of micro-scale models that need to be executed at a macro-scale time step as

\[
\eta(t) = \text{DoF}(1 - g(t)).
\]

To complete the process, the number of processors allocated to run one micro-scale model \( i \), \( P_{\mu i} \), is determined at runtime by the HMC manager. These numbers are stored in an array \( (P_{\mu}) \) to represent the number of processors to run the \( \eta(t) \) micro-scale model(s) for one macro-scale iteration \( t \). Also, the computing resources for the HMC manager \( P_H \) are determined separately.

At compile time, the HMC pattern software obtains the number of processors for each main component (i.e., \( P_M, P_D \) and \( P_H \)) and macro-model degrees of freedom (DoF) from the user. Also, the initial performance measurements of the micro-scale model are benchmarked to retrieve the minimum \( P_{\text{min}} \) and maximum \( P_{\text{max}} \) number of processors to run one micro-scale model. \( P_{\text{min}} \) is determined by memory requirements, and \( P_{\text{max}} \) by strong scaling behaviour where \( P_{\text{max}} \) denotes the number of processors where the execution time is minimised, after which it starts to increase again.

Depending on the budget of the user, the requested time to completion and the expected number of micro-scale models, the user would estimate the resources to be used and the time required for execution. Next, the variables used during runtime, namely the number of processors per micro-scale model \( P_{\mu i} \), must be determined.

Generally, the total number of processors for micro-scale models \( P_{\mu} \) can be used either in stateful or stateless mode. If the micro-scale models are stateful, then the micro-scale models stay in memory and the HMC manager feeds the micro-scale models with appropriate input and initial conditions. Although this method is easy to implement and straightforward, it assumes that all micro-scale models will be the same and take the same time and computational power, which is a main source of a load-imbalance situation.

In stateless mode, the number of processors per micro-scale model \( P_{\mu i} \) is totally dynamic. In this case, the pattern software decides how to use \( P_{\mu} \) for each macro-scale iteration. The main advantage of this method is that when the computing for the micro-scale models is complete,
their nodes/processes can be used for pre-calculating some of the database properties of the surrogate model, or simply release them to the system.

The distribution of the number of processors per micro-scale model depends on the number of micro-scale models requested and the number of available processors for these micro-scale models. In the next subsections, we will illustrate a mathematical model for the dynamic number of micro-scale models and show different cases for the performance of the surrogate model.

**Mathematical model**

In this section, we will address the best way to run the micro-scale jobs. We first have to analyse the computing time for one and for multiple micro-scale models and then evaluate the performance of the surrogate model $g(t)$ per time step during the simulation.

The computing time to run one micro-scale model using $P_{\mu i}$ processors can be calculated utilising the concept of fractional overhead [63, 135], with

$$T^{\mu i}(P_{\mu i}) = \frac{T^{\mu i}(1)}{P_{\mu i}} + T_o(P_{\mu i}) = \frac{T^{\mu i}(1)}{P_{\mu i}}(1 + f_o(P_{\mu i})),$$

where $T^{\mu i}(1)$ is the time to run one micro-scale model using a single processor and $T_o(P_{\mu i})$ is the overhead time for running one micro-scale model using processors. The fractional overhead $f_o$ for running one micro-scale model using $P_{\mu i}$ processors can be expressed as

$$f_o(P_{\mu i}) = \begin{cases} 0, & \text{if } P_{\mu i} = 1 \\ \frac{P_{\mu i}T_o(P_{\mu i})}{T^{\mu i}(1)}, & \text{otherwise.} \end{cases}$$

The total time $T$ to run each micro-scale model on the same number of $P_{\mu i}$ processors (so, independent of $i$) using a total of $P_\mu$ processors for all micro-scale models will be

$$T = \frac{\eta(t)}{\lceil P_\mu / P_{\mu i} \rceil} \left(\frac{T^{\mu i}(1)}{P_{\mu i}}(1 + f_o(P_{\mu i}))\right) \sim \frac{\eta(t)}{P_\mu}(T^{\mu i}(1)(1 + f_o(P_{\mu i}))),$$

where $\eta(t)$ is the number of micro-scale models in time step $t$. The target is to minimise $T$, then

$$P_{\mu i} = 1; f_o(1) = 0,$$

$$T = \frac{\eta(t)T^{\mu i}(1)}{P_{\mu i}}.$$ 

Otherwise,

$$T = \frac{\eta(t)T^{\mu i}(1)}{P_\mu}(1 + f_o(P_{\mu i})),$$

which means that the fewer processors we use per micro-scale model, the lower will be the overhead.
Surrogate model performance

The number of micro-scale models $\eta(t)$ changes with every macro-scale model iteration and is highly dependent on the state of the surrogate model. For this, we need to analyse the performance of the surrogate model per time step ($g(t)$). The value of $g$ can vary for the cases where the user is building the surrogate from scratch ($g \rightarrow 0$), where the surrogate model is replacing the micro-scale models efficiently ($g \rightarrow 1$), or for in-between situations. To deal with the distribution of the number of processors in these cases, we defined three different phases, and the corresponding processor distribution mechanism is shown in Algorithm 1.

**Algorithm 1** HMC phases

1: procedure HMC($g(t), DoF, P_{min}, P_{max}, P_\mu$)
2: $\eta(t) = DoF(1 - g(t))$
3: if $\eta(t)P_{min} > P_\mu$ then  $\triangleright$ Phase 1
4: run($\eta(t), P_{min}$)
5: else if $\eta(t)P_{min} < P_\mu < \eta(t)P_{max}$ then
6: run($\eta(t), \lceil P_\mu/\eta(t) \rceil$)  $\triangleright$ Phase 2
7: else
8: run($\eta(t), P_{max}$)  $\triangleright$ Phase 3
9: end if
10: end procedure

The important elements in Algorithm 1 are the number of processors reserved for all micro-scale models $P_\mu$ and $\eta(t)$, the number of micro-scale models in time step $t$. If $P_\mu < P_{min}\eta(t)$, then the most appropriate action to take is to do farming by running each micro-scale model with a minimal number of processors $P_{min}$. On the other hand, if $P_\mu < P_{max}\eta(t)$, then running with $P_{max}$ is the most suitable choice. Otherwise, for $P_{min}\eta(t) < P_\mu < P_{max}\eta(t)$, all micro-scale models must be run on $\lceil P_\mu/\eta(t) \rceil$, limiting it to not more than $P_{max}$, which is the maximum number of processes that the model can benefit from before the performance decreases. Also, if the performance values of running the micro-scale models using different parameters are available or can be estimated, the number of processors per micro-scale model can be changed accordingly in the second phase. We will apply this concept to two different cases for the surrogate model, starting from scratch and from a developed surrogate model.

**Case (a): Surrogate model from scratch**

If the user starts building the surrogate from scratch, $g \rightarrow 0$, then it is meaningless to run $DoF$ micro-scale models to fill the database at the beginning. This will be inefficient, since the $DoF$ can easily reach $10^6$ or more. What we can do is cluster input parameters into an input subsample. In this case, we first run the initial batch of micro-scale models ($\eta_{init}$), from which we build the surrogate model, and then run the next macro-scale iteration (either look it up in the database or run the micro-scale model), and so on.

Figure 4.3 shows the performance of a surrogate model (top graphs), expressed by $g(t)$ and the corresponding number of micro-scale jobs (lower graphs) for two different performance levels of the surrogate model. Both examples had a $DoF = 48818$. The colours show the three phases, as introduced above. Phase one, where the farming of jobs is done using $P_{min}$ processors
per micro-scale model, is represented in green. Phase two is shown in blue and the third and final phase is shown in red. The dashed lines in the figures are the macro-scale model iterations. The first example, Fig. 4.3(a), shows a surrogate model with good performance, while the second example, Figure 4.3(b), demonstrates poor performance. These performance figures are based on results from a simulation in which this "surrogate model" was actually implemented [133] with modification at the first few macro-scale iterations to mimic the case of a new surrogate model.

![Figure 4.3: Performance of a surrogate model generated from scratch for two different performance levels. Top graphs show the performance values of the surrogate model and lower graphs show the corresponding numbers of micro-scale jobs. In this example, $P_{min} = 1, P_{max} = 16$ and $P_{\mu} = 206$. The colours refer to the phases, where green is the first phase, blue the second, and red the third. The dashed lines represent the macro-scale model iterations.](image)

Case (b): A developed surrogate model

A HMM simulation can also be executing using a previously constructed surrogate model. Figure 4.4 (top) shows the performance of a well-established surrogate model and the corresponding micro-scale models (Fig. 4.4 (lower)) for two different performance levels of the surrogate model. As for case (a), both simulations had a DoF = 48818. The performance figures are based on results from a simulation in which this "surrogate model" was actually implemented [133].
Figure 4.4: Performance of a well-developed surrogate model for two different performance levels. Top graphs show the performance values, and lower graphs show the corresponding numbers of micro-scale jobs. In this example, $P_{\text{min}} = 1, P_{\text{max}} = 16$ and $P_{\mu} = 206$. The colours refer to the phases, where green is the first phase, blue the second, and red the third. The dashed lines represent the macro-scale model iterations.

As shown in Figs 4.3 and 4.4, the first phase of the new surrogate model, case (a), requires more jobs at the beginning to build the surrogate model. This phase also takes more macro-scale iterations to complete. Note that the number of micro-scale jobs per iteration is calculated as $\eta(t) = D_{0}F(1 - g(t))$. However, in the first phase we do not run the total number of 48818 jobs, but we run batches from which we can then train the surrogate model. In the second phase, the number of micro-scale jobs is less than the number of micro-scale jobs in the first phase. Knowing that it is not beneficial to run a micro-scale job utilising more than $P_{\text{max}}$ processors, and the time to run a micro-scale job varies with the number of processors and the input parameters, we might have a number of idle processors. For the two study cases, we can use the free processors in the second and third phases to further explore the parameter space of the micro-scale model for better performance of the surrogate model, or even change the number of processors per micro-scale model based on different input parameters for each micro-scale model. In the third phase, it is preferable to release a number of unused processors back to the system, to save on the budget. Generally, switching between phases will be totally dynamic and the runtime part of the HMC pattern software should handle this process, as will be illustrated in the next sections.

4.3 Red Blood Cell HMM Application

To present the benefit of our scheduling method for HMM applications, we first present our exemplar HMM application. Before doing so, we first need to clarify that in this section, while we describe an outline of the model, we have not yet fully realised the required scale-bridging
algorithms. This is ongoing research and lies outside the scope of this manuscript. However, we do have the main ingredients, the macro-scale and micro-scale solvers available, allowing us to obtain actual execution traces of an HMC run, assuming surrogate performance as introduced in Section 4.2, as will be further clarified below.

Resolving the properties of whole blood on the multiple spatial scales present in the human body is a significant challenge for a single blood-flow model. In this section, we present an HMM model for three-dimensional (3-D) cell-resolved blood-flow in order to address this challenge. The purpose of this model is to resolve whole blood as a suspension of cells with their emerging rheological properties, on the spatial/timescales where continuous models for blood-flow are normally used. A schematic overview of this 3-D HMM model is shown in Fig. 4.5.

![Figure 4.5: The main structure of the red blood cells HMM example used in this section and the different scales (macro, micro and meso).](image)

The 3-D HMM blood-flow model is a macro-scale continuous blood-flow solver that incorporates a local viscosity that depends on the local cell volume fractions ($H$) and shear rate ($\gamma$). The local cell volume fraction of red blood cells, the haematocrit, is described by an advection–diffusion solver on the macro-scale which requires the shear-induced red blood cell diffusivity. This ensures that at every location, a local haematocrit and shear rate will be known. Both the local viscosity and diffusion coefficients required by the macro-scale model are not known and are calculated from cell-resolved micro-scale models for each haematocrit and shear rate combination.

In this application, the blood-flow is modelled on the macro-scale as a continuous fluid using the lattice Boltzmann method (LBM). On the micro-scale, the plasma is modelled by LBM, with a discrete-element method for the red blood cells that are coupled to the fluid via the immersed boundary method [91, 126]. The macro-scale advection–diffusion is also modelled with the LBM. The micro-scale models simulate a perfect shear environment using Lees–Edwards boundary conditions [136] for each haematocrit and shear rate combination. These micro-scale models usually occur in small spatial ($\approx 100\mu$m) and temporal scales ($\approx 1$ms). The micro-scale models
send their local viscosities and diffusivities back to the macro-scale model(s) (spatial scale \( \approx 1 \text{cm} \) and temporal scale \( \approx 1 \text{s} \)) as input for the lattice velocity calculations which, in turn, are used to calculate shear rate. These velocities and diffusivities are used as input for the advection–diffusion solver to produce the density. Therefore, the micro-scale models measure both the dynamic viscosity and diffusion coefficient of red blood cells in this shear environment and return them to the macro-scale model, which ensures that the local viscosity, as well as the haematocrit profiles, on the macro-scale are informed by the cell-resolved models at the micro-scale. Figure 4.5 shows the main structure and the different scales (levels) of the red blood cells HMM example used in this section.

The surrogate model contains a database that is used to store data to avoid duplicating calculations and to build a surrogate model (e.g., based on a Gaussian process) to conduct interpolations for similar parameters. This process decreases the required number of micro-scale models requested, which in turn decreases computation time. However, in this example, we replaced this process with performance figures, which mimic the operation. The performance of the surrogate model was explained in Section 4.2 and the different cases are illustrated in Figs 4.3 and 4.4.

The performance of the surrogate model will vary the number of micro-scale models needed for each macro-scale iteration significantly, which can lead to load imbalance and low utilisation of the available resources. Thus, the main target of the runtime HMC pattern software is to schedule this dynamically varying load of micro-scale models in an optimal way on the available resources.

To understand and improve the scheduling of these micro-scale jobs, we need to profile the average performance of the micro-scale models first. Figure 4.6 shows the execution time of the micro-scale model as a function of the number of processors. In the micro-scale, we simulated the red blood cells and platelets suspensions in plasma using HemoCell [91]. In this specific benchmark, it is clear that the minimal execution time is obtained using 16 processors. Increasing the number of processors to more than 16 actually increases the execution time. This is a well-known phenomenon in strong scaling of parallel applications, and is due to increasing overhead time as the number of processors increases. Thus, in this example, the boundary limits for the number of processors for the micro-scale models are \( P_{\text{min}} = 1 \) and \( P_{\text{max}} = 16 \).

To confirm the choice of farming a large number of micro-scale models using the least number of processors, in addition to the discussion presented in Section 4.2, we will use the following example. Assume that we reserve \( P_\mu = 1024 \) processors for micro-scale models and that in one macro-scale iteration, it was required to run \( \eta(t) = 4096 \) jobs. This is the first phase of the surrogate model performance (i.e., \( \eta(t)P_{\text{min}} > P_\mu \)). By running the simulation using one processor per micro-scale model, we need four batches to finish, where each batch will take \( \sim 3 \text{h} \). This means that a total of \( \sim 12 \) hours of computing is required. On the other hand, if we decide to run 16 processors per micro-scale model, then we would need 64 batches at 30 min each, resulting in a total of \( \sim 32 \text{h} \) of computational time.

4.4 Results

In the following set of experiments, we measured the runtime and utilisation for the four cases of performance shown in Figs 4.3 and 4.4. Resource utilisation \( U \) is defined as \( U = R/C \), where \( R \) is the actual used number of processors and \( C \) is the capacity, which is the total number of processors available for the job. The utilisation was measured as a function of wall clock time during the execution.
In all experiments, the number of processors allocated for the macro-scale model was $P_M = 8$, for the surrogate model $P_D = 1$ and for the HMC manager $P_H = 1$. The total number of processors available for the micro-scale models was $P_\mu = 206$. The degree of freedom selected, for simplicity, was $DoF = 48818$.

We executed our experiments on Eagle\textsuperscript{1}, a supercomputer at the Poznan Supercomputing and Networking Center (PSNC) in Poznan, Poland, which has an Intel Ivy Haswell architecture with a peak performance of about 1372 TFLOP/s. The QCG pilot job used in this work was a normal job submitted to Eagle to reserve a set of resources. After reservation, these resources were then managed using a python script, the Pilot job manager. In this script the user can launch, request and kill jobs dynamically. In our benchmarks, the Pilot job reserves a number of processors first. Then, in the Pilot job manager, we submit the macro-scale model and the HMC manager. The macro-scale, after an iteration, requests a number of parameters from the surrogate model. The surrogate model looks in the database, interpolates the missing quantity and requests to run a number of micro-scale models for the missing quantities. In the benchmarks, we mimic this operation by using the performance of the surrogate model. The number of micro-scale models and the available resources are then sent to the HMC manager to suggest the right distribution of the resources to the Pilot job manager. Also, it acts to different phases of the HMM application accordingly. The pilot job manager then executes the submodels utilising an internal queue on the required resources, gathers the values from the micro-scale jobs and sends them back to the surrogate model.

Figure 4.7 (top panels) presents the utilisation of the system, with assistance from the HMC manager, and the surrogate model starting from scratch. The utilisation of the performance of the surrogate model presented in Fig. 4.3 (a) is shown here in Fig. 4.7a (top panel), while the utilisation of the performance of the surrogate model presented in Fig. 4.3 (b) is shown in Fig. 4.7b (top panel).

In Fig. 4.7a (top panel), in the first macro-scale iteration, a large number of micro-scale jobs are executed, each executing with $P_{min} = 1$ in a first-in, first-out queue (in our method, we use

---

\textsuperscript{1}https://wiki.man.poznan.pl/hpc/index.php/Eagle
Figure 4.7: Utilisation of the system (top panels) by using the surrogate model, as shown in case (a) in Section 4.2. The corresponding number of micro-scale jobs (middle panels) and number of processors per micro-scale job (lower panels) were decided by the HMC manager.

sub-queues in the pilot job rather than in batches). The utilisation in this phase is high because we simply exploit all available resources. At the beginning the utilisation is one, which means that all 206 processors are used as shown in Fig. 4.7a (middle panel). After a while, ∼ 18 − 32 jobs are completed, shown as the first few small decreases of the green line.

The blue line in Fig. 4.7a (top panel), for the second to fifth macro-scale iterations, shows $P_{\min} \eta(t) < P_{\mu} < P_{\max} \eta(t)$ phase. The utilisation at the beginning of these iterations is one, because all the processors reserved for micro-scale models are used by running the micro-scale models with $\eta(t)$, $\lceil P_{\mu} / \eta(t) \rceil$. For example, in the second macro-scale iteration (which lasted from 800 to 1100 minutes of the runtime), 142 jobs were run with ∼ 2 − 1 processors, each shown in Fig. 4.7 (a) the middle and lower panels, respectively. As a result of different micro-scale model execution times with different numbers of processors per micro-scale model invocation, we notice a gradual decrease in the utilisation. In this situation, an internal mechanism could be implemented to use the available processors to refine the surrogate model by proactively (so not informed by the macro-scale model) executing micro-scale models in yet unexplored regions of the micro-scale input parameter space. The gradual decrease in the second macro-scale iteration does not occur in the next iterations (macro-scale iterations 3 to 5), because the number of micro-scale model jobs in iteration 4, for example, is smaller (54 running with ∼ 4 processors shown in blue in Fig. 4.7a (middle and lower panels)).

The last macro-scale iteration in Fig. 4.7a (top panel) falls in the third phase, where the number of micro-scale jobs is only 12, and the number of processors per micro-scale job run is $P_{\max} = 16$. As is shown clearly by the red line in the graph and also the red lines in Fig. 4.7a (middle panel), we run all the micro-scale models in one fast run. This phase is fast, but the utilisation remains low. In this state, as discussed, we could release the unused processors as the surrogate is mature enough to replace the need to generate new micro-scale jobs.
While the second case, Fig. 4.7b (top panel), shows similar behaviour, the surrogate model is not as effective as for the first case. Figure 4.7b (middle panel) illustrates the corresponding number of micro-scale jobs and the average number of processors per micro-scale jobs. Here, we need to generate more micro-scale jobs to sample parameter space at the beginning. This example also shows that after running a large number of micro-scale jobs for the first three macro-scale iterations, we reached a steady level where we ran 35 micro-scale jobs in the second phase and ~2 micro-scale jobs in the third phase utilising a dynamic number of processors.

When a previously developed surrogate model is used, the resulting utilisation of the system is as shown in Fig. 4.8 (top panel), and the corresponding number of micro-scale jobs and the average number of processors per micro-scale jobs are shown in Fig. 4.8 (middle and lower panels, respectively). This case shows the situation where the surrogate model must run a large number of micro-scale models at the beginning to fill the database rapidly and then becomes sufficiently effective to replace the need for micro-scale models.

Figure 4.8: Utilisation of the system (top panels) by using a previously developed model, as shown in study case (b) in Section 4.2. The corresponding number of micro-scale jobs is shown in the middle panel. The lower panel shows the corresponding number of processors per micro-scale job decided by the HMC manager. The colours represent the phases, where green is the first phase, blue the second, and red is the third. The dashed lines represent the macro-scale iterations.

Figure 4.8 (top panels) appears similar to the previous case, but we notice that because the difference in the number of micro-scale jobs requested for each macro-scale step is high, there are more idle processors in the first phase than for the previous case. Here, we might consider returning the processors back to the system up to where the first few macro-scale iterations are completed as shown in Fig. 4.8 (middle panels).

As expected, reaching the steady state in this case was faster than in the previous case, and in the subsequent runs, the surrogate model replaced the need for generating new micro-scale jobs.

Finally, note that in production runs, the number of iterations on the macro-scale will be much larger, and if the surrogate model is performing very well, the overall utilisation can be
very small (as, e.g., reported in ref [133]). Once the HMC run is running steadily in phase 3, the HMC manager should return resources to the system. We have not yet implemented this feature.

4.5 Conclusion

In this contribution, we showed the mechanism of running HMM applications using pattern software. In our experiments, we executed an HMC assuming four different scenarios of surrogate models. In each scenario, we show the number of micro-scale jobs generated at each macro-scale iteration, and the utilisation of our system, using a pilot job and the HMC manager, with the focus on running the micro-scale jobs on the right resources.

We propose that the execution of the micro-scale models in the HMM application should be viewed as three distinct phases: the first phase is when we have a very large number of micro-scale models ($P_{\mu} < \eta(t)P_{\max}$). In this phase, the appropriate action is to conduct farming with $P_{\min}$ processors per micro-scale model. This reduces the overhead, as we demonstrated mathematically and with experiments. The second phase is when ($P_{\max}\eta(t) < P_{\mu} < P_{\max}\eta(t)$). In this phase, we run the required micro-scale jobs with $\lceil P_{\mu}/\eta(t) \rceil$. During the last phase, when the number of micro-scale jobs requested is less than the total number of available processors for micro-scale models, the most appropriate action is to run all the remaining jobs using $P_{\max}$ processors. The surrogate model is nearly or fully developed at this stage and it can replace the need to generate micro-scale jobs efficiently. At this stage, certainly when we need to run many iterations, releasing unused processors will result in an increase in utilisation and a reduction of the computational cost.

Although farming is a well-known method, the act of dynamically changing from one phase to another under the control of a dynamically evolving surrogate model, with its corresponding actions and decisions, is a new approach. In our experiments, we substituted the actual implementation of the Gaussian process regression of the surrogate model with the performance of the surrogate model, as observed in a HMM for predicting material properties [133]. This performance figure provides the required number of micro-scale jobs that are needed at each macro-scale iteration. It remains to be seen if such surrogate model performance is achievable for our blood-flow HMM application.

Finally, we used a pilot job manager and requested a number of processors. The pilot job allows small jobs to run for different resource requirements that are defined dynamically. Thus, our method is totally dynamic, where both the number of micro-scale models and the resource requirements are decided at runtime.
Load Balancing of Parallel Cell-Based Blood-Flow Simulations

Abstract

The non-homogeneous distribution of computational costs is often challenging to handle in highly parallel applications. Using a methodology based on fractional overheads, we studied the fractional load-imbalance overhead in a high performance biofluid simulation aiming to accurately resolve blood flow on a cellular level.

In general, the concentration of particles in such a suspension flow is not homogeneous. Usually, there is a depletion of cells close to walls, and a higher concentration towards the centre of the flow domain. We perform parallel simulations of such suspension flows. The emerging non-homogeneous cell distributions might lead to strong load imbalance, resulting in deterioration of the parallel performance. We formulate a model for the fractional load-imbalance overhead, validate it by measuring this overhead in parallel lattice Boltzmann-based cell-based blood-flow simulations, and compare the arising load imbalance with other sources of overhead, in particular the communication overhead. We find a good agreement between the measurements and our load-imbalance model. We also find that in our test cases, the communication overhead was higher than the load-imbalance overhead. However, for larger systems, we expect load-imbalance overhead to be dominant. Thus, efficient load balancing strategies should be developed.

5.1 Introduction

Load imbalance is the situation where the workload between processes is unequal. In the case of cellular suspension simulations, it might arise due to the inhomogeneity of the cell distribution within the simulation domain. This is typically a major source of loss of parallel efficiency for

---

biofluid flow simulations such as blood-flow simulations. Blood is a dense suspension of red blood cells (RBCs). Typically, the volume fraction of RBCs in human whole blood is 40%-45%. Blood also contains other cells (platelets, white blood cells) with volume fractions of less than 1% [137]. The large number of RBCs present in the blood (5 million/mm$^3$) [138] leads to complex rheological behaviour such as shear thinning [139]. Another interesting phenomenon is the cross-stream migration of the RBCs, which is caused by the RBCs’ complex motion and deformability, and the presence of solid tube walls [140]. This migration results in a higher RBC concentration in the middle of a channel flow and a depletion of RBCs near the channel walls, leading to a cell-free layer (CFL) that acts as a lubricant speeding up the core of the channel flow [140, 141].

For a more detailed understanding of the physiological processes related to cardiovascular disease (such as atherosclerosis, embolisms, thrombosis, and thrombolysis), we should be able to simulate hundreds of millions of individual RBCs and other blood constituents explicitly [98, 111, 142]. This requires extensive computing resources, notably on massively parallel computers [143].

Our cell-based blood-flow model, presented in [98, 144], uses the lattice Boltzmann method (LBM) [34], which is a mesoscale method for solving fluid flows. For this purpose, we used Palabos$^1$, a fully parallelised LBM-based fluid solver available in the public-domain. The immersed boundary method (IBM) [145] is a fluid–structure coupling method used to provide a bidirectional interaction between the fluid and the vertices of the RBCs. The RBCs themselves were modelled using a boundary element method. HemoCell, which is a cell-flow simulation code based on Palabos, was developed earlier and benchmarked [98]. The HemoCell code implements the vertices of RBCs using a spectrin–link model to represent an RBC [146], and also implements a systematic coarse-graining procedure to select a varying number of vertices [147]. Furthermore, the HemoCell code defines models for RBC–plasma and RBC–RBC interactions [98] using Lagrangian and Eulerian descriptions.

Both Palabos and HemoCell utilise a domain decomposition algorithm for parallelisation [98]. Currently, domains are chosen to be equal in volume, i.e., they contain a similar number of Eulerian fluid points in each subdomain. This might create a load imbalance because of the CFL and the elevated RBC concentration at the core of the flow (i.e., the inhomogeneity of the RBC distribution, which is common in suspensions in general). Based on our previous benchmarks, we find that most of the computing time is spent on the RBC dynamics, namely RBC–RBC and RBC–fluid interactions [98]. Therefore, the computing time in a domain is mainly determined by the number of RBCs in that domain. This load imbalance might become a major source of loss of parallel efficiency when a simulation runs on a large number of processors. In addition, execution also suffers from communication overhead, and it is not immediately clear to what extent the load-imbalance overhead is more severe than the communication overhead. We did not address this in earlier performance measurements [98] because in that study the domain was unbounded (no walls present). Clausen et al. [143] reported a decrease in efficiency because of load imbalances. However, to the best of our knowledge, a structured and detailed study, both theoretical and experimental, on the effect of RBC distribution in channel flows on overall performance has not been carried out.

Here, we discuss extensive load-balance studies on suspension flow in channels, using the concept of the fractional load-imbalance overhead proposed by Axner et al. [135] and the fractional communication overheads proposed by Fox [148]. We extend these concepts to

---

$^1$www.palabos.org
suspension simulations, propose a theory to estimate the fractional load-imbalance overheads and validate this theory with measurements.

5.2 Parallel Performance Expressed in Terms of Fractional Overheads

Our starting point was the performance prediction model of Axner et al. [135] and the notion of fractional communication overheads introduced by Fox [148]. First, assume that the execution time of a parallelised simulation on one processor, $T_1$, is equal to the execution time of the sequential simulation:

$$T_1 = T_{seq}$$

We write the parallel execution time on $P$ processors $T_P$ as:

$$T_P = \frac{T_{seq}}{P} + \sum_j T_{\text{Overhead}}^j (P),$$

(5.2)

where $T_{\text{Overhead}}^j$ is the time spent on the different overhead types denoted by $j$. Overheads can include communication, or load imbalances. Axner et al. showed that it is possible to identify many separate sources of overheads. They also showed that overheads can, in principle, be measured and predicted with some degree of accuracy [135].

The speed-up and efficiency can be written as:

$$S_P = \frac{T_1}{T_P} = P / (1 + \sum_j f_j),$$

(5.3)

$$\varepsilon_P = \frac{S_P}{P} = 1 / (1 + \sum_j f_j),$$

(5.4)

where

$$f_j = \frac{T_{\text{Overhead}}^j}{(T_{seq}/P)},$$

(5.5)

are the fractional overheads. Note that, by combining Eq. 5.2 and Eq. 5.5, $T_P = (T_{seq}/P) (1 + \sum_j f_j)$. This means that every overhead $j$ adds a fraction $f_j$ execution time to the ideal execution time $T_{seq}/P$.

Note that if $f_j \ll 1$ for all $j$ overhead types,

$$\varepsilon_p \approx 1 - \sum_j f_j,$$

(5.6)

Expressing different types of overheads $j$ in terms of fractional overhead defined in Eq. 5.5 allows easy comparison of these overheads, as they appear in the equation for the parallel efficiency, and enable us to determine the severity of one overhead compared with another. Note that the overheads depend on the number of processors, not only explicitly, but also through the dependence of $T_{\text{Overhead}}^j$ on $P$. Thus, comparing the overheads as a function of $P$ will give an indication of which overhead should be reduced first to improve the overall scalability of the simulation.

In this study, we mainly focus on finding an expression for the fractional load-imbalance overhead $f_{LI}$ for our cell-based blood-flow simulations, and then compare it with the fractional communication overhead $f_{comm}$. However, this framework can be used for any type of parallel computing. More extended expressions and measurements are provided by Axner et al. [135].
CHAPTER 5. LOAD BALANCING OF PARALLEL CELL-BASED BLOOD-FLOW SIMULATIONS

Fractional load-imbalance overhead

Assume that the communication time is zero and that the time taken by processor $i$ to perform its required computations is $t_i$, where $i = 0, ..., P - 1$. We now assume a load imbalance, so in general, $t_i \neq t_j$. This will lead to a reduction in efficiency because the execution time of parallel code is determined by the slowest processor, i.e., the maximum $t_i$ among processors:

$$T_P = t_i^{\max} = (T_{seq}/P) + [t_i^{\max} - (T_{seq}/P)], \quad (5.7)$$

where we have written the expression in a form matching Eq. 5.2. If there is no load imbalance, i.e., all $t_i$ are equal, then $T_{seq} = Pt_i$ and the second term in Eq. 5.7 equals 0, as required. From Eq. 5.5, the fractional load-imbalance overhead can be expressed as:

$$f_{LI} = [(t_i^{\max} - (T_{seq}/P))/(T_{seq}/P)] = (t_i^{\max}/ < t_i >) - 1, \quad (5.8)$$

where $< t_i >$ is the average computation time per processor.

An interesting feature of Eq. 5.8 is that it also takes heterogeneous architectures into account, and expresses the load imbalances caused by the presence of faster and slower processors in a parallel machine. Load imbalance can arise due to the different workloads per processor, but also by equal computational loads executed on processors with different computational speeds (as can happen in large HPC systems). Eq. 5.8 naturally captures both effects, as it is fully defined in terms of execution times per processor. It also captures the combined effect of unequal amounts of work per processor and unequal processor speeds.

Fractional load-imbalance overhead for RBC suspensions

As discussed by [98], the execution time of the cell-based blood-flow simulation is mainly determined by the computations required for simulating the mechanical behaviour of the RBCs. Take $N$ as the total number of RBCs in the simulation and $\tau$ the required computing time per RBC. This leads to

$$T_{seq} = N \tau n_{iter}, \quad (5.9)$$

where $n_{iter}$ is the total number of iterations (time steps). Note that we assume that the additional time for the Lattice Boltzmann flow computations can be neglected. We also assume that $\tau$ is constant. We can now calculate the time per iteration per processor $i$:

$$t_i = N_i \tau, \quad (5.10)$$

where $N_i$ is the number of RBCs in the domain of processor $i$ and $N = \sum_{i=0}^{P-1} N_i$. In HemoCell, this is not entirely correct, as a single RBC can be in several domains at the same time, thereby introducing small errors as some RBCs could be counted twice if they cross processor boundaries. We formulate the theory in this manner to keep it insightful; however, the expressions can be easily formulated in terms of the number of RBC discretisation points per processor. In general, $N_i \neq N/P$, which leads to a load imbalance. This could, for instance, occur because the subdomains in the domain decomposition are all equal in size but the concentration of RBCs differs in the computational domain (as in blood flow in an artery). Substituting Eq. 5.10 in Eq. 5.8 results in:
PARALLEL PERFORMANCE EXPRESSED IN TERMS OF FRACTIONAL OVERHEADS

$$f_{LI} = (N_i^{\text{max}} \tau / (N \tau / P)) - 1 = (N_i^{\text{max}} / <N>) - 1.$$  \hfill (5.11)

If the number of RBCs is the same in each processor, then \(f_{LI} = 0\), as expected. To calculate \(f_{LI}\) theoretically, we need information on the concentration profile in the flow domain. This may not always be known, but for now we assume that it is available, either through experiments, theory, or earlier simulations. Thus, using the haematocrit profile, the domain size, the volume of RBCs, and knowing the domain decomposition used for the parallel simulation, we can calculate the numbers of RBCs in a processor, \(i\), as follows:

$$N_i = H_i (V_i / V_{\text{RBC}}).$$  \hfill (5.12)

where \(N_i, H_i\) and \(V_i\) are the number of RBCs, the local average haematocrit value and the volume of the domain of processor \(i\), respectively. RBCs conserve their volume, so \(V_{\text{RBC}}\) is constant among the processors. From this, we can find the maximum and mean number of RBCs, and from that the theoretical value of \(f_{LI}\). This value is compared with the actual measurements obtained from the simulations.

Simulation set-up

We considered flow in two types of domains. First, we considered flow between two parallel plates, which has a relatively simple 1-D "averaged" haematocrit profile. Next, we considered flow in a rectangular channel, which gives rise to a more complicated 2-D haematocrit profile. In all cases, we obtained simulation results for three domain sizes, resulting in different systems with different numbers of RBCs ("small" 4290, "intermediate" 7150 and "large" 10868, respectively, all at a constant average haematocrit of 38\%) for both the channel flow and the parallel plates cases. Moreover, we considered three different domain decompositions, as depicted in Fig. 5.1. We considered two types of 1-D decompositions (Fig. 5.1 b,c), which allowed a detailed study of the load imbalance and a 3-D decomposition (Fig. 5.1 d) similar to those used in production runs. The intermediate 2-D decomposition case is not reported since the results are comparable to the 1-D and 3-D cases (data not shown). The number of cores ranged from 1 to 16 in all cases. Moreover, for the largest number of RBCs, we ran the 3-D decomposition using 1 to 1024 cores. All parameters are shown in Table 5.1. The domain size and the volume of RBCs are expressed in lattice units (LU), where in these simulations 1 LU is equal to 1\(\mu\)m. The relative size of the RBCs is 8 LU. The RBCs can cross processor boundaries. As described in detail in Ref.[98], to do so, three different fields (for fluid, surface particles and cell particles) with different envelope sizes are used and need to be transferred over domains.

As Fig. 5.1 shows, the blood flows in the positive x-direction. The channel flow case is a rectangular domain bounded by four no-slip walls parallel to the x–y and x–z planes, whereas the parallel plates case is bounded by two no-slip walls parallel to the x–z plane only. For both cases, a periodic boundary condition is applied on the inlet and outlet of the domain, and for the parallel plates, it is also applied in the z-direction. The Reynolds number is tuned to be equal to the experiment in [143].

We measured the haematocrit profile in our simulations, based on the concentration of the Lagrangian points that construct the RBCs in a unit volume. For a measurement of the computational load this is more accurate than considering the centroids of the RBCs. For the parallel plates case, we computed the average haematocrit profile in the y-direction; for the
Figure 5.1: (a) Sequential domain, with blood flow in the positive x-direction, (b) 1-D decomposition perpendicular to the blood flow (vertical), (c) 1-D decomposition parallel to the blood flow (horizontal) and (d) a 3-D decomposition.

Table 5.1: Parameters used in the simulations, where N denotes the number of RBCs, LU means lattice units (1 LU = 1 μm), and Re is the Reynolds number of the flow.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>&quot;small&quot;</th>
<th>&quot;intermediate&quot;</th>
<th>&quot;large&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry</td>
<td>Channel flow and parallel plates</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain size</td>
<td>126 × 64 × 128 LU³</td>
<td>210 × 64 × 128 LU³</td>
<td>320 × 64 × 128 LU³</td>
</tr>
<tr>
<td>Number of processors</td>
<td>1–16</td>
<td>1–16</td>
<td>1–1024</td>
</tr>
<tr>
<td>Number of red blood cells</td>
<td>4290</td>
<td>7150</td>
<td>10868</td>
</tr>
<tr>
<td>Volume of a red blood cell</td>
<td>91.5077 LU³</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flow type</td>
<td>(1) x–y and x–z planes for channel flow walls parallel to the x–y plane</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2) x–z plane only for the parallel plates</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haematocrit</td>
<td>38%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Re</td>
<td>5.8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

channel flow case, we computed the average profile in the y–z plane. Next, we used these measured haematocrit profiles and the number of processors to compute the fractional load-imbalance overhead, according to Eqs. 5.11 and 5.12.

5.3 Results

To demonstrate the introduced method, we measured the haematocrit profiles for both the channel flow and the parallel plates cases. Next, the theoretical values of the fractional load-imbalance overhead were calculated from the maximum and average numbers of Lagrangian points in the subdomain using (Eq. 5.10). Then, we directly measured the pure computing time per processor \( t_i \) with the Allinea MAP profiler\(^2\), and from that we computed the fractional load-imbalance overhead directly using (Eq. 5.8). Finally, we compared the experimental values with the theory. We also measured communication overheads with the MAP tool, and computed related fractional overheads.

\(^2\)www.allinea.com/products/map
Haematocrit distribution analysis

The haematocrit profiles for "small", "intermediate" and "large" systems for the parallel plates case are shown in Fig. 5.2. We first equilibrated the simulations by running until the local haematocrit converged to a stable profile with fluctuations below the 1% level (averaged over 1000 time steps). For all system sizes, it is clear that the haematocrit is high near the centre of the channel, and there are fewer RBCs close to the wall. These profiles are in qualitative agreement with earlier simulations [111, 141], although some details, such as the geometry used and the shear rate, are different.

![Haematocrit distribution for the parallel plates case with different numbers of RBCs (N), average haematocrit value of H = 38%, distance between plates = 64 μm and Re = 5.8. Green is for "small", blue for "intermediate" and the red for "large" systems.](image)

**Figure 5.2:** Haematocrit distribution for the parallel plates case with different numbers of RBCs (N), average haematocrit value of $H = 38\%$, distance between plates = $64 \, \mu m$ and $Re = 5.8$. Green is for "small", blue for "intermediate" and the red for "large" systems.

The haematocrit for the channel flow is shown in Fig. 5.3. In all RBC cases, the rectangular haematocrit profiles clearly show depletion of RBCs near the domain walls, with pronounced depletion zones in the corners of the domain.

Measuring fractional load-imbalance overheads

The results of the experiments are divided into two categories. The first category includes the results of fractional load-imbalance overheads over the three different domain decompositions and all three system sizes for 1–16 cores. The second category includes the fractional overheads of the largest system size ("large" system) using 1–1024 cores. The Lisa computing facility$^3$ was utilised to obtain the results of the first category, while Cartesius$^4$ was used for the second. Moreover, we ran the same experiments on Sanam$^5$ giving, as expected, similar results for the load-imbalance overhead (data not shown).

---

$^3$userinfo.surfsara.nl/systems/lisa
$^4$userinfo.surfsara.nl/systems/cartesius
$^5$www.top500.org/site/50458
Different domain decompositions

The first set of results shows the fractional load imbalance for the different types of domain decompositions described earlier. In this set, we use a small number of cores (1–16) to show the load imbalance effect on different domain decompositions. Figure 5.4 shows the theoretical and measured fractional load-imbalance overheads for the parallel plates cases with different numbers of RBCs. We measure the fractional load-imbalance overheads for 1000 time steps and report the average and the standard deviation. In the horizontal domain decomposition, the fractional load-imbalance overheads display the same trend for all system sizes. This is attributed to the similarity in haematocrit profiles, as illustrated by Fig. 5.2. Moreover, the noticeable increase in fractional overhead from two to four cores is caused by the fact that cores that hold the central domains have many more RBCs than cores that hold the domains close to the walls.

For the vertical domain decomposition (Fig. 5.4, middle), the fractional load-imbalance overhead is an order of magnitude lower than for the horizontal case. In fact, in theory, one would expect a zero fractional load-imbalance overhead, as the haematocrit profile does not depend on the x-coordinate. However, note that in some cases the size of the flow domain lattice in the x-direction is not exactly divisible by the number of processors; this leads to small load imbalances caused by different domain volumes per processor. To confirm this, the theoretical fractional load-imbalance overhead for the "large" system case is zero, because the length of the domain in the x-direction for this case is 320 LU, which is perfectly distributed among the different numbers of cores. Measurements reveal that we do get a very small load imbalance, which we attribute to small fluctuations in the haematocrit. On the other hand, for "small" and "intermediate" systems, the size of the domain is not exactly divisible by the number of processors, which leads to an unequal distribution of subdomains among the cores. This is immediately noticeable in Fig. 5.4 (middle) as a small fractional load-imbalance overhead, both in theory and measurements. Thus, unlike the horizontal domain decomposition, the haematocrit distribution did not play a significant role in the vertical domain decomposition.

Finally, the 3-D domain decomposition (Fig. 5.4, right) shows the smallest fractional load-imbalance overheads. This can be attributed to the method we use to decompose the domain,
in which we first decompose the x-axis, then the y-axis and finally the z-axis. Therefore, the number of cores on the y-axis was kept as small as possible. This ensures a smaller effect from the inhomogeneous haematocrit distribution.

Similarly, for the channel flow case, Fig. 5.5 shows the fractional load-imbalance overhead for different decompositions. The effect of the extra walls is clearly visible. In the horizontal domain decomposition (Fig. 5.5, left), we note a steep increase in the fractional load-imbalance overhead from two to four cores, almost double the increase of the parallel plates case. On the other hand, the fractional load-imbalance overhead for the vertical domain decomposition is approximately the same as for the parallel plates case, which was expected.
Figure 5.5: Fractional load-imbalance overheads for horizontal (left), vertical (middle) and 3-D (right) domain decompositions for the rectangular channel case. The solid lines represent the theoretical values; the markers-only represent the measured values. The green (triangles), blue (squares) and red (rectangles marks) lines indicate "small", "intermediate" and "large" systems, respectively. Note the difference in scale between cases.

The extra walls are most noticeable in the 3-D domain decomposition. In this case, the fractional load-imbalance overhead increased by a factor two.

Finally, we observe that the measured values are in agreement with the theoretical values, in the sense that we find the general trend, and that our simple model always correctly identifies the order of magnitude of the fractional load-imbalance overhead. In some cases, there is a mismatch between the model and the measurement, e.g., for the large system with 3-D decomposition (see right panel in Fig. 5.5). This can be attributed to fluctuations in RBC count, and to other sources of load imbalance that do not scale linearly with the RBC count per processor (such as the size of the overlap regions to resolve RBCs that cross processor boundaries [98]).

Large number of cores

For this set of experiments, we used the same configurations as those given in the previous section; however, now only with a "large" system and a larger number of cores (1–1024 cores) were used, and we only considered 3-D decompositions. Moreover, we compared the fractional load-imbalance overhead with other measured overheads. In our measurements, we considered the communication and load-imbalance overheads. Figure 5.6(a) illustrates the fractional overheads for the parallel plates case. Here, the communication overhead is dominant. The load imbalance caused by the existence of external walls is observed when the number of cores in the y-direction is increased. For example, the increase in fractional load-imbalance overhead from 16 cores $(4 \times 2 \times 2)$ to $32 \ (4 \times 4 \times 2)$ is due to doubling the number of cores in the y-dimension from two to four.

Note that the total combined overheads (i.e., sum of the fractional communication and the fractional load-imbalance overhead) can be as large as 1, or even higher. The case of 1 for instance means that with respect to the ideal situation with zero overheads the execution time is doubled. The communication overhead in Fig. 5.6 was measured using the Allinea Map profiler. These measurements are substituted in Eq. 5.5, assuming $T_{seq} = T_1$. The source of the communication
overhead is mainly due to transmitting data between processor boundaries (related to the fluid flow field and RBCs crossing processor boundaries [98]).

As before, the fractional load-imbalance overhead for the channel flow, as shown in Fig. 5.6, is higher than for the parallel plates case. The communication pattern is less dominant in this case, and, for a small number of cores, the fractional communication overheads are comparable to the fractional load-imbalance overheads.

In general, the fractional communication overhead shows the same order of magnitude as the fractional load-imbalance overhead and an increase is observed in both cases. We observe an increase in the communication overhead in the channel flow case, due to the fact that communication depends linearly on the haematocrit. Higher haematocrit in the middle of the channel raises both fractional communication and load-imbalance overheads.

Experiments have been carried out on both Lisa and Cartesius. Comparing Figs 5.4, 5.5 and 5.6 shows, for up to 16 processors, they have similar fractional load-imbalance overheads.

5.4 Discussion

We introduced fractional load-imbalance overhead for parallel suspension simulations, specifically cell-based blood-flow simulations. We formulated a methodology to estimate the fractional load-imbalance overhead based on haematocrit profiles, and compared it to direct run-time measurements, demonstrating good agreement. We used the haematocrit profile because of
its importance in the blood flow. These calculations could, however, be generalised to any suspension simulation.

In the 3-D domain decomposition, the fractional load-imbalance overhead was smaller than the fractional communication overhead. However, for larger system sizes (i.e., for orders of magnitude more RBCs), we expect a decrease in the fractional communication overhead while the fractional load-imbalance overhead remains the same. Given the size of the fractional load-imbalance overheads on $1024$ processors, $f_{LI} \approx 0.5$, the best achievable efficiency in this set-up would be $2/3$ (see Eq. 5.6). Add to that the communication overheads, and it is clear that for productions runs, one needs to achieve better load balancing by reducing the fractional load-imbalance overhead with at least one order of magnitude.

In this work we mainly concentrated on differences in local haematocrit due to the presence of solid walls. However, there are also inhomogeneities possible in unbounded flows, caused by structure formation such as clustering [54]. So, if processor domains become smaller than typical cluster sizes, that could also lead to load imbalance. However, as these clusters are very dynamic objects, we would expect that this would lead to an increase in fluctuations in the fractional load-imbalance overhead.

For future work, we will develop a load-balance library (LBL), which will reduce the imbalance situation, and in turn will lead to a decrease in the fractional overheads and an increase in efficiency. This will be shown in the next chapter. The LBL could achieve load balance by adaptive non-equal domain decomposition, by tuning the clock speed of individual processors or by a combination of both. Taking the clock speed adjustments into account will add an energy reduction factor into the equation, in addition to the targeted efficiency, where in the end the reduction of wall clock speed is the only target. We plan to rely on METIS, with haematocrit profiles as input. After the initial phase of the simulation, this profile might change. Therefore, rebalancing of the load should take place, but only when the rebalancing overhead is much lower than that due to the load imbalance itself and the fractional load imbalance is more than a given (problem-specific) threshold value.
Abstract

We report on our approach towards load balancing parallel cell-resolved blood-flow simulations, considering load imbalance caused by flow boundaries and the underlying numerical lattice as well as that caused by time-dependent cell-concentration profiles. We rely on our theoretical framework for fractional overheads to predict load-imbalance overheads in a benchmark problem, the static flow of red blood cells and platelets in a straight cylindrical channel. We validate our theoretical framework for this benchmark by comparing predictions with actual measurements of fractional load-imbalance overheads and also compare these with other overheads, most notably for communication. For load balancing itself, we rely on the building cube method, in which the computational domain is subdivided into small atomic blocks, which are then assigned to processors so that the computational load is balanced while communication is minimised. This is achieved by applying the Kernighan–Lin graph-partitioning algorithm available in METIS. Parameters were derived from the simulations and were quantified to obtain the maximum parallel efficiency with minimum fractional overheads. These parameters include quantifying the lowest fractional load-imbalance overhead possible, a weighting function in which fluid nodes and particle nodes are used individually and in combination to determine computational load, and the atomic block size, which determines not only the fractional load-imbalance overhead but also the communication overhead. The results demonstrate the effectiveness of this load balancing for varying numbers of processors and atomic blocks. Finally, we consider the case of a time-dependent cell-concentration profile where we measured both the fractional load-imbalance and fractional communication overheads during two phases, initial equilibration and final equilibrium. Dynamic load balancing was applied which increased the parallel efficiency by 10%.

6.1 Introduction

Blood primarily consists of red blood cells (RBCs), platelets and white blood cells, suspended in plasma. The main functions of blood include delivering oxygen from the lungs to the tissues in the body and transporting waste products. It also assists in safeguarding the body against parasites and infections and is instrumental in the healing process after injuries. From a macroscopic perspective, blood in large arteries can be modelled as a homogeneous fluid. However, blood is actually a suspension of deformable cells within a viscous plasma fluid. This deformability, in addition to the large volume fraction of RBCs in blood (some 40%), leads to complex behaviour, such as shear thinning and formation of a cell-free layer (CFL) [126, 149–151].

To study RBC-related diseases (e.g., malaria or sickle cell anaemia [138, 152]) and cardiovascular diseases (such as the formation of thrombosis in aneurysms [111, 153–155]), many frameworks have been implemented to simulate blood flow, either as a continuous fluid (e.g., HemeLB [156] and Musubi [157]) or as a fully cell-resolved model (e.g., HemoCell\(^1\), a High pErformance fraMewOrk for dense CELLular suspension flows [91, 126]). In these examples, the lattice Boltzmann method (LBM) is applied to model flow, which depends on a stream-collision scheme between neighbouring points on a specific lattice [158–160]. Many other models have also been used for continuous and cell-based blood-flow simulations, e.g., relying on solving the Navier–Stokes equations [161] or the dissipative particle dynamics method [162]. Here, we focus on applying LBM for simulating the plasma, a boundary element method for the deformable RBCs and platelets, and the immersed boundary method for the fluid–structure interaction [126].

The LBM can be relatively easily parallelised because it requires only nearest-neighbour information [163]. In non-trivial sparse geometries, efficient parallel LBM simulations are more complicated but can be achieved by using indirect-addressing implementations of LBM and more advanced load-balancing schemes [135]. Groen et al. [164] also noted that the main challenge of scaling LBM simulations is balancing the computational load effectively. In their work, they illustrated that LBM parallel execution can address non-sparse bulk flow mechanisms efficiently [164]. Schornbaum and Rüde [165] argued that LBM demonstrates exceptional scalability on present supercomputing systems and has, therefore, become a substitute mechanism for large-scale flow simulations.

When we add suspended particles to the fluid model, new sources of load imbalance arise [166]. Such load imbalance in cell-based blood-flow simulations generally occurs because of a non-uniform cell distribution, where close to walls a cell-free layers forms with very low cell densities, while in the centre of the flow channel, the density of RBCs is usually larger than the average density [126, 166]. In addition, as with continuous flow simulations, the geometry of the flow domain can play a significant role in load imbalance. Knowing that the entire process for resolving the blood flow and the cell mechanics is computationally expensive, load imbalance may significantly affect the efficiency of the entire parallel execution, as was noted by, for example, Claussen et al. [143].

Load balancing for LBM is usually achieved by utilising some variant of the building cube method [167], in which the computational domain is subdivided into small atomic blocks, which are then assigned to processors so that the load is balanced while communication is minimised. In general, the fluid nodes should be equally distributed among processors. However, in fully resolved blood-flow simulations, the complex motion of RBCs makes this balancing far from trivial [166], and the minimal size of the atomic block is usually determined by the largest possible

---

\(^1\)https://www.hemocell.eu/
diameter of a deformed RBC. The atomic blocks make up a connected graph in which the nodes can be weighted by the amount of computation per atomic block, and the edges by the amount of communication between the atomic blocks. Load balancing, while minimising the communication, then becomes a graph-partitioning problem, which is well understood and frequently applied in load-balancing scenarios. The space-filling curve and multilevel algorithms are frequently used to ensure load balance.

Several open-source graph-partitioning software packages, such as (Par)METIS and Zoltan are available. Here, we rely on METIS. We use the K-way partitioning technique, which requires three processes. First, the size of the geometry is decreased to the minimum size required for decomposition, known as graph coarsening. Next, the decomposition is performed using some specified algorithm (e.g., the Kernighan–Lin algorithm) and finally, the geometry is refined again to the original size (graph refining). To judge the effectiveness of the decomposition, the edge-cut and balance ratios are estimated.

In the previous chapter, we pointed out that the concentration profile of suspended RBCs in flows is in general not uniform and may even change over time. Because a large portion of the computational time is taken up by the simulation of the cell mechanics and the immersed boundary method, which scales linearly with the number of RBCs, this can lead to a significant load imbalance. We analysed this effect both theoretically and by measuring it in a few well-controlled benchmarks. Here, we go one step further and capture the sources of loss in parallel efficiency of cell-suspension simulations in terms of fractional overheads and show how to minimise the load imbalance.

The availability of heterogeneous architectures is expected to gain in importance, certainly for high-end parallel computers. The results of Galindo et al. imply that a load imbalance can also occur easily between heterogeneous architectures. This can be solved by establishing a strong link between the architecture and the corresponding parallel code. The issue of heterogeneous architectures can also be captured in our theoretical formalism for the fractional load-imbalance overhead, which can also capture the difference in computational speed between processors. However, in this work, we exclusively focus on load balancing on homogeneous computational resources.

In our benchmarks we use HemoCell, a framework for cell-suspensions that is constructed on top of Palabos, a versatile and fully parallelised LBM production code. Both HemoCell and Palabos are available as open-source codes. HemoCell utilises a coarse-grained spectrin-link model to represent a single RBC. It models RBC–plasma interactions using an immersed boundary method. The plasma, the fluid component, is modelled with the LBM as implemented in Palabos.

In our previous work, we formulated a theoretical framework for load imbalance in parallel cell-based blood-flow simulations, utilising HemoCell as a case study. We continue that work here and report on methods to address, quantify, and decrease the fractional load-imbalance overhead, while at the same time also minimising the fractional communication overhead, thus optimising the parallel efficiency. We predict and measure the fractional load-imbalance overhead and identify key parameters that determine its value. We also analyse the situation in which the cell-concentration profile changes over time, to reveal to what extent and when dynamic load balancing is required. The objective is to distribute the load between processors as evenly as possible.
possible and maintain low communication, which will enable the execution of such cell-resolved blood-flow simulations on petaflop (and emerging exaflop) machines.

6.2 Methods

In this section we will briefly summarise the concept of fractional overheads for parallel efficiency, and specifically for load imbalance and apply this to include RBC suspensions. Then, we will illustrate the load-balancing methods and weighting functions used in this chapter.

Fractional overheads

We apply our framework of fractional overheads [166] to characterise the performance of a parallel computation, in which the parallel efficiency is expressed as

\[ \varepsilon = \frac{1}{1 + \sum_i f_i}, \]

(6.1)

where \( f_i \) is the fractional overhead related to some source of overhead \( i \) that reduces the parallel efficiency, such as, e.g., communication. The fractional overhead can in general be expressed as

\[ f_i = \frac{p T_i}{T_{seq}}, \]

(6.2)

where \( T_i \) is the execution time caused by overhead \( i \), \( p \) is the number of processors and \( T_{seq} \) is the sequential execution time of the simulation. For this framework to be useful, it must be possible to actually measure \( T_{seq} \), which may not always be achievable. In our case, \( T_{seq} \) is measured by executing the benchmark simulation using one processor in a dedicated "fat" node with sufficient memory (512 GB/processor).

We can also express load imbalance in terms of a fractional overhead [166]:

\[ f_{LI} = \left( \frac{t_{max}}{< t_i >} - 1 \right), \]

(6.3)

where \( t_i \) is the pure computation time on processor \( i \) excluding all overheads, \( t_{max} \) is the maximum \( t_i \) over all processors and \( < t_i > \) is the average. For parallel cell-based blood-flow simulations, under the assumption that most of the computational time is for the cell mechanics and on homogeneous resources, this can be expressed as [166]:

\[ f_{LI} = \left( \frac{N_{max}}{< N >} - 1 \right), \]

(6.4)

where \( N_{max} \) is the maximum number of RBCs in a processor and \( < N > \) is the average number of RBCs per processor.

Load balancing

In this section, we introduce the process of load balancing cell-resolved blood-flow simulations as shown in Fig. 6.1. For simplicity, we illustrate the process in a 2-D rectangular domain. For more details we refer to B. Initially, the flow domain is voxelised and the atomic block structure is initialised (Fig. 6.1 a and b). This structure is built by specifying either the size or the number of atomic blocks. In the first approach, atomic blocks are created by linearly
partitioning the domain length on each axis of the voxelised domain. These atomic blocks are then distributed amongst the available processors equally (Fig. 6.1 c). The load-balance process starts by measuring the fractional load-imbalance overhead using the measured compute time per processor, Eq.6.3. We measured the fractional load-imbalance overhead during execution using a profiling tool, in our case the Allinea map tool\textsuperscript{4}. Utilising this tool, we also measured the fractional communication overhead. For communication, we quantified the fractional overheads resulting from communication between processors ($f_{\text{comm}}$) and between atomic blocks residing on the same processor ($f_{\text{blocks}}$)\textsuperscript{5}.

If the fractional load-imbalance overhead is larger than a user-defined threshold $f_{\text{Limit}}$, a load balancing will be performed. For this, a weighted graph is constructed from the computational domain. For each atomic block, we extract the number of lattice points (Eulerian points), the number of immersed RBC vertices (Lagrangian points) or a combination of these points. We define three different weighting functions to estimate the computational cost: the number of fluid nodes in an atomic block ($F$), the number of particle nodes in an atomic block ($P$) or a weighted combination of both $P$ and $F$. The weights express the difference in computational cost for a Eulerian point and a Lagrangian point. We used a ratio of 0.02/9.8 for the Eulerian/Lagrangian

\footnote{www.allinea.com/products/map}

\footnote{Palabos implements the building cube method by relying on communication between the atomic blocks, even when they are on the same processor.}
cost ratio, based on a measurement of distribution of compute times between the Eulerian points (the fluid simulation) and the Lagrangian points (the mechanical RBC model and the immersed boundary method).

The numbers collected are used to construct a weighted graph, where the nodes represent the atomic blocks, weighted by an estimate of the computational cost of that block, based on its number of points. The edges are the number of communication points between atomic blocks. The weighted graph is used as input for one of the ParMETIS graph-partitioning functions (ParMETIS_V3_PartKway and ParMETIS_V3_PartGeomKway). The functions used are based on a K-way partitioning technique to suggest the optimal distribution of blocks per processor.

We employ three distinct load-balancing methods. The first (computation-balanced) is optimised to distribute only the computational load equally without considering communication. The second method does the opposite, a partition resulting in minimising communication, under the constraint that all the work cannot be assigned to one processor only (communication-balanced). Finally, the last method attempts to distribute the computation equally while at the same time minimising the communication (both computation- and communication-balanced). The result of this step is a balanced distribution of atomic blocks over processors based on the chosen weighting function and balancing method.

Finally, we assign each atomic block to its processor, delete the old lattice and re-initiate a new one with the new block distribution (Fig. 6.1 d). The communication time between atomic blocks can be high, because of a serialisation process to prepare for the actual send/receive. We therefore merge constitutive atomic blocks inside one processor to larger rectangular blocks within one processor, to decrease the number of blocks per processor (Fig. 6.1 e) and thus to minimise the communication overhead.

### 6.3 Simulation Set-up

We use a straight cylindrical channel flow domain for our simulations, as shown in Fig. 6.2. Note that the circular cross-section in combination with the Cartesian lattice used by the LBM can lead to load imbalance, even without suspended cells. The domain has a diameter of 0.1 mm and a length of 0.25 mm. The spatial resolution of the LBM lattice is 0.5 µm, resulting in 33.5 million lattice sites. The average volume fraction of cells is 45%, which results in a total of 16,278 RBCs. In the current implementation of HemoCell, each red blood cell has 642 vertices, 1920 edges and 1280 faces with different material forces acting on them. In total, 4000 equations are integrated forward in time for every single RBC at every iteration [126]. We ran the simulations with a constant pressure gradient and periodic boundary conditions in the flow direction, and applied the lattice Bhatnagar–Gross–Krook collision operator, bounce-back wall boundary conditions, and the D3Q19 LBM model.

We started our simulation with homogeneously distributed RBCs as initial condition [126]. Next, the cell-concentration profile developed during the equilibration period, which required about 12 million iterations of the LBM. After this period, a fully developed and stabilised cell-concentration profile was reached, and most load balancing experiments that we report are based on this equilibrated profile. However, in the final experiment, we measure the actual development of the fractional load-imbalance overhead during the equilibration period itself.

In the following experiments, we will first quantify the fractional load-imbalance overhead resulting from the underlying numerical lattice only (by setting the cell concentration to 0%, so pure fluid only). This overhead was measured for the three different load-balancing methods
introduced in Section 6.2 (compute-balanced, communication-balanced and both) using the number of fluid nodes as the weighting function ($F$). In this experiment, we obtain the loss of efficiency caused by load imbalance due to the geometry only, based on fluid alone. Even if we could achieve a perfect load balance of the RBC computations, this geometry-induced load imbalance will still lead to a (small) overall load imbalance.

Next, choosing the appropriate weighting function from ($F$, $P$ and both $P$ and $F$), is critical for the load-balancing process. Thus, we illustrate the fractional load-imbalance overhead resulting from each weighting function and compare these overheads with the previous case of pure fluid without cells and a case without load balancing applied (so, using the initial block decomposition as shown in Fig. 6.1 c).

We also study the influence of the size of the atomic blocks as used in the building cube method by varying the number of atomic blocks used in the load-balancing process. The size of the atomic block determines not only the fractional load-imbalance overhead but also the communication penalty, and we need to find the right balance between both types of overheads.

Finally, we measured the time-dependent fractional overheads of our cell-suspension simulation during the equilibration period. We also demonstrate a proof-of-concept dynamic load-balancing process.

We executed our experiments on SuperMUC\(^6\), a supercomputer at the Leibniz-RechenZentrum (LRZ) in Garching, Germany, which has an Intel Ivy Bridge architecture with a peak performance

\(^6\text{www.lrz.de/services/compute/supermuc}\)
of about 3190 TFLOP/s. HemoCell was compiled with GCC-6.2, OpenMPI 1.4.8 and HDF-1.8 for I/O. We ran each experiment to an equilibrium checkpoint and then obtained an average result over 500 iterations for 100,000 time steps after equilibration. We also used the Sanam\textsuperscript{7} supercomputer at King Abdulaziz City for Science and Technology (KACST) in Saudi Arabia for additional runs with the same configurations, for verification. As expected, the results on Sanam for fractional load-imbalance overhead are the same as on SuperMUC, while the fractional communication overheads between both machines are different due to their different architectures (data not shown).

6.4 Results

We present results for two general cases. The first is the static one, where the cell-concentration profile is fully developed and independent of time. The second is the dynamic load-balancing case, in which the cell-concentration profile changes over time.

Static load balancing

We first consider a fully developed and static cell-concentration profile, which for our test case is presented in Fig. 6.3 (left), with the related velocity profile in Fig. 6.3 (right). As illustrated in Fig. 6.3 (left), the centre of the channel has more RBCs than the remainder of the channel. In addition, a cell-free layer is visible, as there are no RBCs near the vessel wall.

Figure 6.3: Cell-concentration profile (left) and the longitudinal flow velocity (right) for the benchmark in Fig. 6.2.

Figure 6.4 shows the fractional load-imbalance $f_{LI}$ and communication overheads between processors $f_{comm}$ and between blocks $f_{block}$ for the straight cylindrical channel flow without cells (cell concentration is 0%) as a function of the number of processors (ranging from 32 to 128), for three load-balancing methods (computation-balanced, communication-balanced and both). In this experiment, we use the number of fluid nodes ($F$) as the weighting function and a constant block size ($\approx 8 \times 8 \times 8$) LBM units. The fractional load-imbalance overhead was both measured using Eq. 6.3 and estimated using Eq. 6.4, where $N$ now represents the number of fluid nodes.

\textsuperscript{7}www.top500.org/site/50458
Figure 6.4: Top panel: measured (solid lines) and estimated (dotted lines) fractional load-imbalance overheads without cells, using three load balancing methods, communication-balance (black), computation-balance (green) and balancing both communication and computation (blue). Middle panel: fractional communication overhead caused by communication between processors (o marker) and between blocks (x marker) for all three methods. Lower panel: the sum of the fractional load-imbalance overhead and both fractional communication overheads.

Figure 6.4 shows a steady increase in the fractional load-imbalance overhead as the number of processors increases. The computation-balanced case shows much less fractional load-imbalance overhead than the communication-balanced case, as expected. The results for the last case (both communication- and computation-balanced) fall between these two lines, being only slightly larger than the pure computation-balanced case. In this case of a pure fluid, there is good agreement between the actual measurements and the theoretically estimated values based on the graph decomposition data.

The fractional communication overheads are shown in the middle panel of Fig. 6.4. We measured both the communication between processors ($f_{\text{comm}}$ marked with circles) and the communication between blocks within a processor ($f_{\text{block}}$ marked with crosses). As the number of processors increased, $f_{\text{block}}$ decreased. This is expected because by increasing the number of processors, we decrease the number of atomic blocks per processor and therefore the between-block overhead. Moreover, the fractional communication overhead between processors for the computation-balanced case is larger than that for the other cases. This can be attributed to non-location-based graph-partitioning decisions. These decisions may lead to assigning non-contiguous blocks to one processor, which adds arbitrary communication points. The
other two scenarios have smaller fractional communication overheads and do not depend very strongly on the number of processors in the range that we measured (unlike the fractional load-imbalance overhead). The fractional communication overhead between blocks shows no significant difference between the three load-balancing scenarios.

The lower panel in Fig. 6.4 shows the total fractional overhead (the summation of the fractional load-imbalance and both fractional communication overheads). We find that over the whole range of number of processors, the total fractional overhead for the third decomposition strategy (both computation- and communication-balanced) is lower than for the other cases, while the total sums for the other two cases are similar. For a small number of processors (e.g., 32), the total fractional overhead for the third decomposition case is about 0.22, which results in a parallel efficiency of 81.6%. For 128 processors, the efficiency decreases to approximately 71.5%. It must be noted that we did not consider overheads from I/O in these experiments. In subsequent experiments we will use the combined balancing method (i.e., both communication- and computation-balanced method).

To choose the most appropriate weighting function, we analysed the fractional load-imbalance overhead $f_{LI}$ in cell-based blood-flow simulations utilising the three weighting functions proposed earlier ($P, F$, both $P$ and $F$) on a normal cell concentration ($H = 45\%$, where $H$ denotes the cell concentration, also called haematocrit for RBCs). To complete the picture, we show the fractional load-imbalance overhead for the same system, but before the load-balance process is applied (no load balance, NLB, only a block-wise decomposition strategy is applied, Fig. 6.1 c) and lastly for a system with no cells ($H = 0\%$), where the only applicable weighting function is the number of fluid nodes ($F$). Figure 6.5 shows the measured (Eq. 6.3) and the estimated (Eq. 6.4) fractional load-imbalance overheads for different processor counts (512, 1024 and 2048). As block size we used $14^3$ LBM units (which is larger than what was used in the experiment reported in Fig. 6.4, due to the constraint that it should fit at least one RBC).

For the three numbers of processors illustrated, the fractional load-imbalance overhead for the NLB cases is the largest and has the highest fluctuations, clearly demonstrating the need to balance the load. The efficiency of 512 processors in the NLB case can drop to 62.5% due to load imbalance alone. This loss in efficiency can become worse as the number of processors increases if it is not resolved.

On the other hand, without cells ($H = 0\%$), the fractional load-imbalance overhead drops to 0.2. It is clear that the fractional load-imbalance overhead in Fig. 6.5 ($H = 0, F$) is higher than for the cases of Fig. 6.4 because of the larger numbers of processors used and larger block sizes (for the same processor count). $f_{LI}$ is still high and can be overcome by using more blocks. In this experiment, the block size used is $14^3$ lattice Boltzmann units because of the assumption that one RBC should not stretch to three or more blocks. However, when we do not have cells ($H = 0\%$), we can decrease the atomic block size to $8^3$ (inset in Fig. 6.5 shows this for 512 processors), to have the same block size as the previous figure (Fig. 6.4). In Fig. 6.5 (inset), it is notable that by adding more blocks (i.e., using smaller block sizes), a gradual decrease in the fractional load-imbalance overhead is achieved. However, this decrease comes at a cost, as the communication penalty increases.

Next we analyse the fractional load-imbalance overhead for the three weighting functions ($P, F$, both $P$ and $F$). We find that the smallest overhead, in general, was obtained using both the number of vertices of the RBCs and the number of fluid nodes (i.e., the $P$ and $F$ weighting function). However, the difference in average fractional load-imbalance overhead between the $P&F$ weighting function and the weighting function using the number of RBCs only is very small,
RESULTS

Figure 6.5: Measured (bars) and estimated (green dots) fractional load-imbalance overheads for the three weighting functions (the number of fluid nodes (F), the number of particle nodes (P) or a weighted combination of both P and F) with a normal cell concentration (H = 45%). Three cases are reported (H = 45,F; H = 45,P and H = 45,P&F). No load balance (NLB) is the same system (H = 45%) but before the load-balancing process. A system without cells (H = 0%), where the only applicable weighting function is the number of fluid nodes (F) is shown as well (H = 0, F). Results are shown for 512 (left), 1024 (middle) and 2048 (right) processors. The error bars denote the range of the smallest to largest measured fractional load-imbalance overheads over 100,000 iterations in the simulation, and the bars denote the average. The atomic block size is $14^3$ lattice Boltzmann units. In the inset, the fractional load-imbalance overhead is measured utilising smaller block sizes ($14^3 - 8^3$) for the no cells case (H = 0%) running on 512 processors.

and taking into account the fluctuations, not significant. Thus, in the subsequent experiments, the number of vertices of the RBCs (P) will be considered as the weighting function, as this reflects the main culprit of load imbalance, the haematocrit profile.

After choosing the balancing method and the weighting function, we estimate the optimal atomic block size. The choice of atomic block size depends on the total sum of the fractional overheads, which in our case are the fractional load-imbalance overhead and the fractional communication overheads. In Fig. 6.6, we measured (solid lines) and estimated (dotted lines) the fractional load-imbalance and measured the fractional communication overheads both between processors ($f_{\text{comm}}$ marked with circles) and between-block overheads ($f_{\text{block}}$ marked with crosses). This was done for a set of block sizes for the same number of processors as was used before (512, 1024 and 2048), after utilising the number of particles as a weighting function.
CHAPTER 6. LOAD BALANCING MASSIVELY PARALLEL CELL-BASED BLOOD-FLOW SIMULATIONS

Figure 6.6: Measured (solid lines) and estimated (dotted lines) fractional load-imbalance overheads and measured fractional communication overheads between processors (o marker) and between blocks (x marker), using multiple block sizes for three processor counts, 512 (top panel), 1024 (middle panel) and 2048 (lower panel). On the horizontal axes we show the atomic block size and in brackets the average number of atomic blocks per processor.

For all three processor counts, the measured fractional load-imbalance overhead is generally in agreement with the estimated one. We observe that the fractional load-imbalance overhead decreases with decreasing block size (i.e., with increasing number of blocks), as expected. The fractional communication overhead between processors is comparable with the fractional load-imbalance overhead for smaller numbers of blocks per processor (i.e., large block sizes), but the difference between these fractional overheads increases gradually. In HemoCell, by increasing the number of blocks, we increase the number of communication neighbours and therefore we need more time for communication. The fractional communication overhead between blocks increases slightly with the block size but does not have a significant effect on the overall performance.

With 512 processors (shown in the top panel of Fig. 6.6), for example, the best fractional load-imbalance overhead is achieved when utilising a 14^3 atomic block size. However, with this small block size, communication becomes the bottleneck and the efficiency drops significantly to approximately 30%. The least total fractional overhead was achieved by utilising a block size of 60 x 13 x 7, which means \approx 1.8 blocks per processor and similar results for the other processor counts. The results, therefore, show that keeping the number of blocks per processor between 2 and 5 would reach the best performance, as illustrated in Fig. 6.6. To guarantee utilising the favourable number of blocks, while ensuring a minimal fractional load-imbalance overhead, we should use small atomic blocks first and then merge the local atomic blocks to a few larger blocks. We should also further optimise the inter-block communication overhead.
Dynamic load balancing

Finally, we ran our system (with $H = 45\%$) from a homogeneously distributed initial state through an equilibration period to an equilibrium state of the cell-concentration profile, measuring the fractional load-imbalance overhead $f_{LI}$ and the communication overheads $f_{comm}$ and $f_{block}$ as a function of simulated time. At the start of the simulation, we performed a domain decomposition as before. In this experiment, we utilised $\approx 3$ blocks per processor, with 3168 blocks in total and we executed the experiment on 1024 processors. In Fig. 6.7, we show the measured fractional load-imbalance overhead (green line) and the fractional communication overheads between blocks (lower blue line) and between processors (top blue line) as a function of time. Then, after $\sim 16\mu s$, we rebalance the load of the domain using the same methods as before.

![Figure 6.7](image)

**Figure 6.7:** Measured (green line) fractional load-imbalance overheads and fractional communication between processors (top blue line) and between blocks (lower blue line). Development over time, using 1024 processors and block size of $21^3$. Two separate phases are shown: initial equilibration and final equilibrium separated by the small red bars. The vertical red line denotes the time of the applied load-balancing step.

The time-dependent fractional load-imbalance overhead in Fig. 6.7 suggests that there are two different phases in the equilibration. During the initial equilibration phase (from $0\mu s$ up to the first red marker), the distribution of RBCs does not really change and the cells slowly increase their velocity. The fractional load-imbalance overhead remains low (about 0.1). Between the first and the second red markers, we observe a gradual increase in the fractional load-imbalance overhead. In this period the RBCs stretch due to the emerging shear-stresses. This is the period where the concentration profile slowly changes from a flat distribution to the equilibrated distribution from Fig. 6.3. Finally, $f_{LI}$ reaches a plateau of $\approx 0.65$, which is the final equilibration of the cell-concentration profile (quasi-stationary). At this stage, a rebalancing of the load was performed to avoid performance losses (at the red line in Fig. 6.7). The fractional load-imbalance overhead decreased to $\approx 0.45$, and that fractional load-imbalance overhead was maintained for the last $\approx 3$ million iterations. This overhead is consistent with what we measured earlier for the same problem in Fig. 6.5 running on the same number of processors (1024) using a $(H = 45, P)$ weighting function.
As shown in Fig. 6.7, both communication overheads were small in the beginning and slowly increased. After the load-balance step, the overhead of communication between processors decreased, which might be because of obtaining a semi-equal communication between processors or deleting temporary communications variables, thus reducing the synchronisation time between processors. In our test case, the efficiency dropped to 40% because of load-imbalance and communication overheads. By doing a load balance, the efficiency increased to 50%. Finally, we measured the wall clock time required for the rebalancing of the load and re-initialisation of the numerical lattice (together we call this rebalancing) in HemoCell. We found that the time to rebalance is 65 seconds, which is equivalent to 400 iterations. So, the fractional rebalance overhead, $f_{RB}$, can be calculated using

$$f_{RB} = P(t_x/n_{RB}), \quad (6.5)$$

where $P$ is the number of processors, $t_x$ is the equivalent number of iterations to rebalance and $n_{RB}$ is the number of iterations until the load balance is performed. In this example, we balanced the load after $n_{RB} = 16$ million iterations and the rebalance process takes the time of $t_x = 400$ iterations. Thus, the fractional rebalance overhead is 0.026, which is negligible to the decrease in the other overheads due to the rebalancing ($\sim 0.3$), so the cost of rebalancing is well invested.

6.5 Conclusions

We have identified sources of load imbalance in a cell-based straight cylindrical channel flow and quantified this fractional load-imbalance overhead from the concentration profile of a cell-suspension simulation. We noticed that the inhomogeneous distribution of cells in a straight cylindrical channel geometry would result in a different workload distribution from that for a simple fluid. Depending on how the amount of computation for the cells compared with that for the fluid, the uneven distribution of the workload could decrease the efficiency significantly. We also quantified the communication overheads, both between blocks and between processors, as fractional communication overheads, allowing us to compare these quantities directly with the fractional load-imbalance overhead. The main target is then to reduce these overheads using constrained optimisations.

The fractional load-imbalance overhead can cost as much as 40% of the efficiency in some cases. Even without cells, this loss can be 10%. We observe that when we decrease the block size, we obtain better load balance, but the communication might also increase. We investigated three balancing methods (communication-balance, computation-balance and balancing both communication and computation). By using the fractional overheads, we illustrated that the best load balance was achieved by balancing both computation and communication. Also, we tested different weighting methods (using the number of Lagrangian points, number of fluid nodes and a combination of the two). The best load balance was obtained by using a combination of the number of Lagrangian points and the number of Eulerian points. However, in our implementation and benchmarks, we chose to balance the load based on the Lagrangian points due to a higher computation demand of these vertices compared with the fluid nodes, and because in these benchmarks, we update the particle material model at every single step of the fluid. In more decoupled runs, when the particle material model is updated every 20 fluid steps, the choice of a preferred weighting function will be shifted more towards considering both the fluid and the particle nodes. The fractional communication overhead was comparable with the fractional load-imbalance overhead using
3–5 blocks per processor. This quantity is code dependent, but it is important to maintain this number to assure high efficiency.

We investigated two phases of the time-dependent cell-concentration profile, equilibration and final equilibrium. In the initial equilibration, where the distribution of cells does not significantly change and the velocity of cells is slowly increasing, the fractional load-imbalance overhead remains low. Then, the cell-concentration profile slowly changes from the initial distribution to the equilibrated distribution. In this part of equilibration phase, the fractional load-imbalance overhead increases and finally reaches a nearly steady value.

In future work, predicting the load balance based on cell-concentration profile predictions is an important step. We believe that if we can predict the local cell-concentration profile, then we can also predict the fractional load-imbalance and possibly also communication overheads, and, thus, perform the load-balance procedure as a more deeply integrated step in the execution pipeline. Quantifying how effective this is will be a topic of future investigations.
Conclusions and Future Work

In this thesis, the concept of multiscale computing patterns is presented and, through partially implementing it, a proof of concept has been provided. These computing patterns can be seen as a high-level call sequence of single-scale models representing a functional decomposition of multiscale models. The multiscale applications are categorised based on their common "patterns". After detection of a specific pattern, i.e., a generic call sequence of single-scale models, specific algorithms can be used to ensure highly parallel computing efficiencies. For example, this can be applied to load balancing. Although it is beyond the context of this study, the impact of these computing patterns on assisting with check-pointing mechanisms and energy consumption can also be investigated.

Three patterns were identified and illustrated using generic task graphs. Based on our studies, we believe that these three patterns, in addition to their combinations, are comprehensive and most of the multiscale applications would fall into one of these three categories. Extreme scaling, the most generic pattern, comprises a set of multiscale applications that has one or more primary submodel(s) and a set of auxiliary submodel(s). In this pattern, scheduling these submodels to the appropriate HPC computing resources and the execution elements is an important step. Failing to do so would typically result in idle cores. Moreover, reducing the interface communication between the primary and auxiliary submodels to a minimum is beneficial. In replica computing, one macro-scale model is coupled to a large set of independent micro-scale models and each only has one pre-processing step. Thus, the replicas could benefit from being distributed to different supercomputers. Heterogeneous multiscale computing shares a similar architecture to that of RC, but the main differences are the inclusion of surrogate models which result in dynamic execution patterns of the micro-scale models. In HMC, a large and dynamic number of micro-scale models per iteration can lead to significant load-imbalance issues, especially when a normal and contentious reservation of cores/nodes is used without ending and re-submitting the job. This can also lead to long queueing times. Both RC and HMC have such dynamic features, whereas HMC is tightly coupled.

The concepts proposed in this study for multiscale computing patterns are partially realised in Multiscale Computing Patterns software and a few examples are demonstrated. We maintain the separation of concerns used in MMSF. Three different layers constitute the Multiscale Computing Patterns software, namely a description of the multiscale application, optimisation criteria and execution components. In the Description component, the computational requirements and the task graph of the single-scale models are defined. These definitions are then used as input to
the Optimisation component, which suggests where to run the single-scale models based on the objectives of the user-chosen and pattern-specific optimisation criteria. This produces a set of proposed execution plans with their expected execution times to the Execution component. The Execution component then predicts the total time to completion by adding the predicted queuing times and then selecting the shortest time plan.

Performance modelling of HPC applications, both analytical and simulation, is currently researched very actively. These models help predict application runtime, and potentially other resource usages, to help increase the efficiency of job submission and execution. The parameters in performance models allow users to estimate performance at scales previously not possible, or on new machines. Therefore, they must account for the performance characteristics of the underlying system (CPUs, memory system, interconnects, etc.), and their correlation with the application performance. However, the resulting performance models are highly specific and tuned to the specific parameter set for which they were designed, and extensive testing and validation are often required.

When discussing performance models for multiscale models, another dimension to the problem exists. The coupling between each individual single-scale model has an impact on the performance of the overall simulation. The nature of the single-scale coupling may be either synchronous or asynchronous, depending on whether the communicating model must wait for a response before progressing with its own calculation. Thus, the performance of both the coupling and the associated models must be a factor in the performance model, and this is what we are aiming to optimise, specifically between the primary and auxiliaries in the extreme scaling pattern. However, in other cases, such as in replica-exchange simulations, where submodels communicate with each other, the coupling should be considered as an external factor to the performance model.

The performance of single-scale models is an important factor and affects the overall multiscale performance significantly. In this thesis, we analysed and quantified the load-imbalance status in cell-suspension simulations using the notion of fractional overhead. A new methodology to calculate the fractional load-imbalance, \( f_{LI} \), and communication overheads, is presented. Starting with a concentration profile and knowing that most of the computation is spent on the suspended particles, we could use the values of \( f_{LI} \) to decide when to perform repartitioning. Moreover, for the blood-flow simulation case studies used in this thesis, predicting the cell concentration will contribute towards predicting the load imbalance, as well as when to rebalance.

Synchronisation and communication between the single-scale models within a multiscale simulation can happen in a variety of ways, such as direct communication (e.g., TCP sockets) or output files in a directory. These methods will have their own associated costs, such as network latency or file I/O bandwidth, and these factors must be accounted for when constructing a coupled performance model.

A further factor to consider when modelling multiscale simulations is that of job distribution. If the simulation is distributed across multiple scheduler jobs then there is a synchronisation cost associated with the queueing time for both concurrent and successive jobs. Without an advanced scheduler reservation, job start times are beholden to the scheduler, and these start times are very hard to predict in advance.

From the results achieved in this study, we can conclude that multiscale applications facilitate simulations of complex phenomena in less time and with less computing, compared with monolithic codes. However, constructing a multiscale application with adequate computational resources and scale-bridging mechanisms is far from trivial and prone to errors. We found many
variations of multiscale applications that could fit into one or more of the proposed computing patterns, or a combination of them. The computing architectures and requirements also add to the complexity of the problem. For example, after benchmarking a multiscale application, the Multiscale Computing Patterns software decided to run a multiscale application using 256 cores on a specific machine, but this machine did not allow jobs below 1024 cores to run for more than five minutes. As a result, the Multiscale Computing Patterns software selected the second-best choice, which was another machine with a longer queueing time. It is exactly this kind of scheduling decision that we want to hide from the users, where the pattern software, in combination with execution environments such as the Experimental Execution Environment in the ComPat project\(^1\)—whereby multiple HPC systems were coupled and accessible for the multiscale simulations—allows us to do so.

The appropriate computing certificates, e.g., the grid certificate, can be a burden to users because they are not clearly documented and are not unified among systems. In the Experimental Execution Environment of the ComPat project, we implemented a unified workstation where a user can submit from one machine (QCG at Poznan, Poland) to get to all the resources of three different production HPC machines (SuperMUC at LRZ, Garching, Germany; Eagle at PSNC, Poznan, Poland; and Neil at STFC, Daresbury, UK). However, this is generally not the case. The user needs to obtain different certificates for different grids and run jobs, depending on the relevant policies. It would be good to see user communities adopt the concept of such coupled execution environments.

Hybrid computing patterns are clear contenders for continuing this work. For uncertainty quantification, for example, we would need several replicas for an extreme scaling application (Fig. 7.1). In this process, we would run several instances of the same multiscale application with different parameters in \(A^+_1\) (parameter sweeping) to find optimal values in \(A_2\) and to quantify the error generated from the different parameters.

![Figure 7.1: Generic task graph of one example of the proposed hybrid multiscale computing patterns. Here, a combination of two patterns is observed. An ES multiscale application, represented by primary model \(A\), serial auxiliaries \(B_S\) and parallel auxiliaries \(B_P\), is shown in the dashed boxes, and the replica computing is represented by the dynamicity feature of \(A^+_1\) and the post-processing step in \(A_2\).](image)

The concept of multiscale computing patterns clearly is not rigid and can easily be extended to include other possible computational patterns, which were not considered in this study. For example, classifying other patterns based on very detailed computing energy consumption, or classifying multiscale applications based on memory usage, could be more beneficial for energy

\(^1\)http://www.compat-project.eu/
consumption optimisation. In particular, energy consumption will be an important metric in the future and the number of cores used per hour could well be replaced with joules used per hour. In considering the energy consumption of multiscale applications, one would not only consider the energy consumption of single-scale models and the communication between them, but also that of the architecture used (e.g., the clock frequency, memory usage, etc.).

Finally, predicting the fractional load-imbalance overhead could be one more step towards a complete dynamic and beneficial framework to address the (dynamic) load imbalance before it occurs. To achieve this, further in-depth studies should focus on quantifying how much the system could benefit from this process, and how accurate the results would be.
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


Listing A.1: Submodel definitions (matrix.xml): this template shows how to define a single-scale model. Resource name and nodeType specify the resource name in the EEE and the type of the node (see Table 3.1) in this resource. The complete submodels definition is explained in 3.4.
Listing A.2: Multiscale coupling file (multiscale.xml); the template contains input requirements needed to run the multiscale application, such as minimum and maximum number of cores required by the user to run a simulation, instances, coupling topologies and middleware-specific requirements. The complete multiscale coupling description is explained in Section 3.4.

```xml
<performance>
  ...
  <instance id="">
    <benchmark/>
  </benchmark>
  <iterations/>
  <benchmark/>
  <resources name="">
    <nodeType/>
    <numberOfCores/>
    <wallClockTime/>
  </resources>
  </instance>
</performance>
```

Listing A.3: Template of the performance section (in matrix.xml), which is, as described in 3.3, an output of the Translation component. The complete performance section is explained in section 3.4.

```xml
<kernels>
  <kernel/>
  <helper id="/">
  </helper>
</kernels>

<classes>
  <class id="/">
  <node host="/"></node>
  </class>
</classes>

<plans>
  <plan id="/">
    <criteria>
      <time/></time>
    </criteria>
    <group>
      <kernel refid="/">
        <class refid="/">
          <cores></cores>
        </class>
        </kernel>
        </group>
        </plan>
</plans>
```

Listing A.4: Template of Pattern-Driven Planner output. The first part addresses the kernels and helper references. The second provides the references for the classes of nodes. The last part is the set of allocation plans. A plan is a group of single-scale models (which can be the whole multiscale model or part of it) with
a number of classes of nodes and cores assigned to each submodel in a group. Note that the chosen cost criterion is time-to-completion. This file is input to the Execution component. The complete plan structure is explained in Section 3.3.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<qcgJob>
  <task persistent="true" taskId="">
    <requirements>
      <patternTopology>
        <kernels>
          <kernel id="">
            <helper id="" />
          </kernel>
        </kernels>
        <classes>
          <class id="c1">
            <node host="supermuc">thin</node>
            <node host="eagle">haswell_128</node>
          </class>
          <class id="c2">
            <node host="supermuc">thin</node>
          </class>
        </classes>
        <plans>
          <plan id="plan1">
            <criteria>
              <time>T0H10M</time>
            </criteria>
            <group>
              <kernel refid="">
                <class refid="c1">
                  <cores>1024</cores>
                </class>
              </kernel>
            </group>
          </plan>
          <plan id="plan2">
            ...
          </plan>
        </plans>
        <reservations>
          <reservation host="eagle">eagle-res-1</reservation>
          <reservation host="supermuc">srv04-ib.44.r</reservation>
        </reservations>
        <execution type="compat">
          <executable>
            <application name="muscle2" version="compat-1.2" />
          </executable>
          ...
        </execution>
      </execution>
    </task>
  </qcgJob>
```

**Listing A.5:** A set of QCG plans to run one of the three suggested plans using the pattern topology of QCG. For more information on QCG middleware, please see [120].
In this appendix we provide some additional information on the dynamic load-balancing algorithms that we have used in our experiments. As Algorithm 2 illustrates, Palabos starts by restructuring the numerical lattice into a number of atomic blocks by loading the original lattice structure and the block size desired (Lx, Ly and Lz) in the `initAtomicBlocks()` function. Then, after $t_{balance}$ iterations of the simulation in HemoCell, the fractional load-imbalance overhead is measured using the `measureFractionalLoadImbalance()` function, which uses the compute time per processor (Eq. 6.3). However, if the measurement tools are not available, then we `estimateFractionalLoadImbalance`, relying on Eq. 6.4. Finally, when the fractional load-imbalance overhead is larger than a user-defined maximum $f_{Limit}$, the `doLoadBalance()` function is called, where a weighted graph is constructed and used as input to a ParMETIS partitioning function. Note that Algorithm 2 applies also at the start of the simulation, so at the first time step the load balancing is performed.

Algorithm 2 Main structure of the HemoCell simulation with load-balance functions

```
1: procedure main(originalAtomicBlockStructure, Lx, Ly, Lz)
2:  initAtomicBlocks(originalAtomicBlockStructure, Lx, Ly, Lz)
3:  ...
4:  if iter $\% t_{balance}$ then
5:      if measurement is available then
6:         if measureFractionalLoadImbalance() > $f_{Limit}$ then
7:            doLoadBalance()
8:         end if
9:      else
10:         if estimateFractionalLoadImbalance() > $f_{Limit}$ then
11:            doLoadBalance()
12:         end if
13:      end if
14:  end if
15: end procedure
```
At the beginning, the `initAtomicBlocks` (Algorithm 3) starts by partitioning the x-, y- and z-spaces linearly, using the `linearRepartition()` procedure, and then adds blocks constitutively while we ensure that all the old domain cells are covered.

**Algorithm 3** Atomic blocks initialisation

```plaintext
1:  procedure initAtomicBlocks(ORIGINAL_ATOMIC_BLOCK_STRUCTURE, Lx, Ly, Lz)
2:    x0, x1, y0, y1, z0, z1 ← BoundingBox(ORIGINAL_ATOMIC_BLOCK_STRUCTURE)
3:    rangesX ← linearRepartition(x0, x1, Lx)
4:    rangesY ← linearRepartition(y0, y1, Ly)
5:    rangesZ ← linearRepartition(z0, z1, Lz)
6:    for each block do
7:      intersections ← intersectBetween(block, ORIGINAL_ATOMIC_BLOCK_STRUCTURE)
8:      if block covers domain then
9:        newAtomicBlockStructure.addAtomicBlock(block)
10:       else
11:         newAtomicBlockStructure.addAtomicBlock(intersections)
12:       end if
13:    end for
14:  return newAtomicBlockStructure
15:  end procedure

16:  procedure linearRepartition(x0, x1, wishedLength)
17:    Size ← x1 - x0 + 1
18:    nBlocks ← max(1, Size/wishedLength)
19:    basicLength ← totalSize/nBlocks
20:    for each block id i do
21:      cLength ← basicLength
22:      if i < totalSize%nBlocks then
23:        cLength +=
24:      end if
25:      ranges[i] ← (x0 + cPos, x0 + cPos + cLength - 1)
26:    end for
27:  return ranges
28:  end procedure
```

The function in Algorithm 4 gathers the *measured* compute time per processor, then Eq. 6.3 is used to calculate the fractional load-imbalance overhead. If the measurement tools are not available, then we *estimate* the fractional load-imbalance overhead. This estimation is done by gathering the number of points, based on the weighting function used (number of Eulerian points, number of Lagrangian points or a combination) for each processor and estimating the fractional load-imbalance overhead ($f_{LI}$) using Eq. 6.4.
Algorithm 4 Measure fractional load-imbalance overhead

1: procedure measureFractionalLoadImbalance(...) 
2: \( T \leftarrow \text{get\textunderscore time}(...) \) \hfill \triangleright \text{get compute time per processor} 
3: \( \text{avg} \leftarrow \text{average}(T) \) 
4: \( \text{max} \leftarrow \text{max\textunderscore Value}(T) \) 
5: \( f_{LI} \leftarrow (\text{max}/\text{avg}) - 1 \) 
   \quad \text{return } f_{LI} 
6: end procedure

In \text{doLoadBalance()} function, for each block, we construct the nodes and edges of the connected graph, shown in Algorithm 5, as requirements for the ParMETIS functions (ParMETIS\_V3\_PartK-way and ParMETIS\_V3\_PartGeomKway). The used function returns an array containing the processor i.d. for each atomic block.

Algorithm 5 Load balance

1: procedure doLoadBalance(...) 
2:   for each block do 
3:     \( \text{vwgt[]} \leftarrow \text{atomicBlockPoints}(\text{block}) \) \hfill \triangleright \text{using weighting function} 
4:   end for 
5:   part[] \leftarrow \text{ParMETIS\_V3\_XXX}(...\text{vwgt}..) \hfill \triangleright \text{New atomic blocks distribution per processors} 
6:   newAttribution \leftarrow \text{newAtomicBlockThreadAttribution}(\text{part}) 
7:   create new lattice with newAttribution 
8:   delete original lattice 
   \quad \text{return} 
9: end procedure

The last steps in Algorithm 5 assign each block to the desired processor, create new thread attributions, re-initialise a new lattice with the new block distribution and delete the old one. Finally, the \text{doRestructure()} function, which is not shown, merges constitutive blocks, when possible adhering the rectangular block shapes constrains, in one processor.
Summary

Multiscale computing involves the modelling, characterisation and simulation of complex phenomena on several interacting spaces and/or timescales. We simulate multiscale models by executing multiple coupled pieces of code, each representing a single-scale model. In this thesis, we specifically aimed at multiscale simulations on Tier-0 high performance computing systems, which we call high performance multiscale computing. Based on the Multiscale Modelling and Simulation Framework, we first grouped and analysed various multiscale applications to identify a set of generic multiscale computing patterns. At a high level, we can visualise multiscale computing patterns as generic task graphs comprising call sequences for single-scale models, each of which represents a component of a multiscale model. By identifying a particular multiscale computing pattern and using generic algorithms at the level of the pattern, it should be easier to realise load-balanced, fault-tolerant and energy-aware multiscale applications on high performance computers. A proof of concept is presented in this thesis. The software inputs include the computational requirements and restrictions that are defined for both the single- and multiscale models. The inputs also include optimisation criteria, which consist of user-defined criteria, such as resource efficiencies and pattern-specific criteria, including alternating between two modelling cases. We implemented Multiscale Computing Patterns software to minimise developer-related computational complications and to optimise performance based on the pattern of the multiscale application and the chosen criteria. Although optimising a multiscale application itself is important, particularly in terms of communicating between single-scale elements and finding optimal resources, we first need to optimise the single-scale elements. For the purposes of this study, we analysed the loss in parallel efficiency caused by load imbalance in single-scale parallel cell-based blood-flow simulations, for which we used the HemoCell framework for blood flow. We then quantified this loss in terms of the fractional load-imbalance and communication overheads.
Samenvatting

Multi-schaal rekenwerk omvat het modelleren, karakteriseren en simuleren van complexe verschijnselen op verschillende lengte- en tijdschalen. We simuleren multi-schaal modellen door verschillende modellen op een enkele schaal aan elkaar te koppelen. Ons doel in dit proefschrift is het maken van multi-schaal simulaties op niveau 0 supercomputers, dit noemen we multi-schaal rekenen met goede prestaties. Om een set van gerelateerde multi-schaal reken patronen te identificeren hebben we als eerste verschillende multi-schaal programma's gegroepeerd en geanalyseerd gebaseerd op het "Multiscale Modelling and Simulation Framework". We kunnen multi-schale reken patronen op een hoog niveau visualiseren als gerelateerde stroomgrafieken die bestaan uit modellen op een enkele schaal, die elke weer een component van het multi-schaal model representeren. In dit proefschrift passen we deze theorietoe op een model in de praktijk. De invoer voor het door ons ontwikkelde programma bevat de rekenkrachtvereisten en restricties voor zowel het enkel- als multi-schaalmodel. De invoer bevat ook optimalisatietoepassingen, die gedefinieerd zijn door de gebruiker, zoals de efficiëntie van de hulpbronnen, en patroon-specifieke criteria, bijvoorbeeld de wisseling tussen twee componenten. De patroon-detectie wordt automatisch gedaan. Het multi-schaal rekenpatroon programma is door ons geïmplementeerd om programmer gerelateerde rekenfouten te minimaliseren en om de prestatie te optimaliseren gebaseerd op de multi-schaal kenmerken van de componenten en de gekozen criteria. Het optimaliseren van een multi-schaal model is op zichzelf al belangrijk, in het bijzonder voor de communicatie tussen de elementen en voor het vinden van de optimale hulpbronnen, echter hiervoorkent eerst de individuele componenten geoptimaliseerd worden. In dit proefschrift analyseren we de invloed van ongelijke werkverdeling op de parallele efficiëntie door middel van een geparallelliseerd, cel-gebaseerd simulatieprogramma voor bloedstromingen genaamd HemoCell. Het verlies in efficiëntie kwantificeren we in termen van fractionele ongelijke werkverdeling en communicatiekosten.
تشمل الحواسيب متعددة المقاييس كلاً من النذجة، التوصيف، ومحاكاة التظاهر المعقدة على عدة مقاييس متكاملة وآول
زمانية، محاكاة هذه النذجة تم من خلال تشغيل عدد من التدريبات البرمجية المرتبطة بعضها البعض، بحيث يمثل كل
برنامج فيها تمدج لقياس معين. في هذه الاتجاه سيكون تركيبنا متعمق على محاكاة هذه النذجة باستخدام النظامة
الحواسيب ذات الذاكر العالي (الصف-0)، وهو ما نسمي بالحاسوبية متعددة المقاييس العالية الأداء. لقد قمنا استناداً
إلى إطار النذجة متعددة المقاييس ومحاكاة، تصنيف وتحليل العديد من التطبيقات الخاصة بتقنية متعددة المقاييس
من أجل تحديد مجموعة من الانتقالات العامة والمشتركة بين هذه التطبيقات. بمجرد هذا الضوء، يمكننا وبشكل أشمل
تحديد بعض المشكلات المتعلقة بتقنية متعددة المقاييس مثل موازنة التحكم الحسابي، التنسيق الآتي مع الخلل الحاسوي
وحفظ الطرق الكهربائية المرتبطة بالحاسوبية. في هذه الاتجاه، نحن نقدم إثباتاً لهذا المفهوم عن طريق تطبيق مفترض
المصدر، تحتوي مدخلات هذا التطبيق على المتطلبات الحاسوبية وبعض القيود الموجودة على النذجة الفردية أو النذج
المتعدد المقاييس. بالإضافة إلى ذلك، فإن المدخلات تحتوي على معايير تحسين الأداء التي يمكن للمستخدم إدخالها
للحصول على تجربة أفضل للمصادف الحاسوبية أو معايير خاصة بخط معين للعدد من معدل التواصل بين المصالح ذات تقليد
الحواسيب. لقد قمنا تصميم هذا البرنامج للحد من التقديرات الحسابية ولتحسين أداء هذه النذج استناداً إلى تطبيق
متعدد المقاييس والمعايير المختارة. على الرغم من أن تطوير النذج متعددة المقاييس بعد أمر مهم، وخاصة فيما يتعلق
بالتواصل بين العناصر الفردية الاتجاه وإيجاد الطرق المثل، فإن تصميمها يطلب درياً تجريبية منتعامل بين المقاييس
الخاصة، لذلك يجب علينا مسامحين أداء النذجة ذات المقياس الواحد أو لا. من أجل هذه الدراسة، لقد قمنا تحليل
وتحديد مصادف الفقدان في الكفاءة الحاسوبية النسبية من طرف التوزيع غير العادي للجهد الحسوي في حسابات دفع
المست摻ة إلى خليه واحدة موانعية (الذي تم برمجه وتطوريه). بعد ذلك، قمنا تحديد معالج الفقدان في الكفاءة
الحواسيبية عن طريق تحليل الزيادة في التواصل وخلال في التوزيع بين الحواسيب وتحويلها إلى إرقام داخل معادلات
رياضية.
List of Publications


  - **Contribution:** Hoekstra conceived the research, all authors formalised the vision of Multiscale Computing Patterns, Alowayyed developed the example of the extreme scaling pattern. Alowayyed wrote the drafted manuscript, all other authors contributed in further editing the manuscript.


  - **Contribution:** Hoekstra conceived the research, Alowayyed carried out the research, Závodsky and Azizi provided input on the HemoCell production code and cell resolved simulations. Alowayyed wrote the manuscript, all other authors contributed in further editing of the manuscript.


  - **Contribution:** Hoekstra conceived the research, which was carried out in equal proportions by Alowayyed, Piontek, Suter, Hoenen, Groen, Luk, Bosak, Kopta, Perks, Brabazon, Jancauskas. Alowayyed wrote the manuscript, with input from all authors, who also contributed in further editing of the manuscript.


  - **Contribution:** Hoekstra conceived the research, Alowayyed carried out the research, Závodsky and Azizi provided input on the HemoCell production code and cell resolved simulations. Alowayyed wrote the manuscript, all other authors contributed in further editing of the manuscript.
LIST OF PUBLICATIONS

Co-author

  
  – **Contribution:** Alowayyed contributed in the performance experiments and partially helped in analysing the strong and weak scaling measurements.

  
  – **Contribution:** Alowayyed drafted and revised the MMSF part in the manuscript.

  
  – **Contribution:** Alowayyed had a minor contribution in setting up performance experiments.
Acknowledgements

First of all, I would like to thank my supervisor Prof. Alfons Hoekstra. This work would not see the light of day without his guidance. I consider myself lucky to have worked with him. He is a true leader, positive and encouraging. I really appreciate his support especially in the last few days of submitting this manuscript. I am grateful to Prof. Peter Sloot for the valuable comments and assistance in this script. Likewise, I would like to thank Prof. Peter Coveney for his help and fruitful conversations, shukran jazelaan. I will never forget to thank the PhD committee for the effort in assessing this work.

To my beautiful and small family, my princess Razan and Najd. Razan, thank you very much for being you and supporting me whatever the situation may be! Little Jojo, thank you for letting me work :). No words can ever fully explain this.

To my paramenfen Gabor and Victor. Gabor, you have added a "special soul" to the group. I will never forget how "zavaro" HotS is. Also, thanks to you and Orsi for filling the summer with nice ideas. Victor, thank you for being an awesome programmer in the group - you are a really good addition and support to any group.

To my CSL colleagues: working with all of you was a pleasure! 01010111 01101111 01110100 01101001 01100101 01110100 00100000 01100001 01101100 01101100 00100000 01101111 01100110 00100000 01100001 00100000 01110000 01101100 01100101 01100001 01110011 01110101 01110010 01100101. To Britt, thank you for "dutch"-ing my experience, e.g., Sinterklaas. To Rick for noticing the speed of time, to Ben for coffee talks, to Ioannis for having nice political chat, to Grace for last-day notices, to Visnu for being a Sunday mate, to Anna for being a nice mate to Jojo, to Niels for administrative work abstraction, to Erik for positivity. I will also not forget Mike, Emiliano, Rob, Raymond, Jaap, Drona, Valeria, Debraj, Jurjen, Marcel, David, Sumit, Lourens, Amir, Tahera, Eva, Cees, Lampos, Eric and Junior. Thank you, friends!

أود أن أتقدم بالشكر الجزيل للمهندس عبد العزيز أحمد، وحثه من الأعمال والتقارير لدعمهم لي في جميع مراحل دراستي الأكاديمية ولولا هذا الدعم المواصل لم يحضر هذا العمل النور، وشكرًا على دعم شباب الوطن في كافة المجالات وشكرًا لإعطائي هذه الفرصة الخدمة وطنيًا.
My stay here in the Netherlands would not be easy without you, Angelique. You are a true friend and an “ideal” neighbor :). We really liked our small chats, dinners and football matches. Also, I will not forget to mention the awesome neighbors who helped us a lot, to Vlad and Drago for the nice food and awesome talks :, to Arnold for his great help in everything and to Sonia and Michala for the awesome and unforgettable king’s day. To Jaap, thank you - without your talented brain, I would not find this neighbor.

Special thanks to my co-author and colleagues from the ComPat project. To Oliver and Onnie for the nice accommodation in Garching, to Tomek for the warm hospitality in Poznan and nice food, to Manuela for administrative help. Thanks also to Derek, James, Maxime, Robin, Piotr, Bartek, David, Simon, Keeran, Olly, Vytautas and Hugh - it was my pleasure to work with all of you.

To Prof. Muner Al-Otaibi; your and your team's support at the Saudi Cultural Mission was very valuable. Your personal support to Saudi scientists, at different levels, is admirable. To Lieke thank you for all of your help. To Prof. Ibrahim, my first manager and teacher, I have learnt a lot from you regarding work, life and tips to success.

To my Arab and Saudi friends, to Prof. Artoli for the initial support to join this awesome group, to Ali for nice go-outs, to Yasser for being there, to Thamer for the encouraging words, to Dr. Tareg for the start-up advice and the nice Sudanese community, to Munther for being my best friend. To my Iraqi friends, Ali, Athraa and family - thank you for the amazing nights we had together. You were a great host and the best, especially in the “Dolama” :).

Finally, I am sorry if I have forgotten anyone, even those who have contributed the least, in this acknowledgment. Thanks to everyone who helped me even with a smile. The PhD journey is an unforgettable experience, and with these few lines written, a new journey will begin.