

A modelling framework for simulating large geographical systems of agents and fields

Kor de Jong

A MODELLING FRAMEWORK FOR SIMULATING LARGE GEOGRAPHICAL SYSTEMS OF AGENTS AND FIELDS

EEN MODELLEERRAAMWERK VOOR HET SIMULEREN VAN GROTE GEOGRAFISCHE SYSTEMEN VAN OBJECTEN EN VELDEN

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit Utrecht
op gezag van de rector magnificus, prof. dr. H.R.B.M. Kummeling,
ingevolge het besluit van het college voor promoties
in het openbaar te verdedigen op

dinsdag 10 mei 2022 des middags te 4.15 uur

door

Kornelis de Jong

geboren op 17 april 1971 te Oosterwolde

PROMOTOREN:

Prof. dr. D. Karssenberg

Prof. dr. M.J. van Kreveld

COPROMOTOR:

Dr. D. Panja

Utrecht Studies in Earth Sciences 252

**A MODELLING FRAMEWORK FOR SIMULATING LARGE
GEOGRAPHICAL SYSTEMS OF AGENTS AND FIELDS**

Kor de Jong

PROMOTOREN:

Prof. dr. D. Karssenberg
Prof. dr. M.J. van Kreveld

COPROMOTOR:

Dr. D. Panja

EXAMINATION COMMITTEE:

Prof. dr. S.M. de Jong, Utrecht University
Dr. H. Kaiser, Louisiana State University, United States of America
Prof. dr. G. Keller, Utrecht University
Prof. dr. N. Meratnia, Eindhoven University of Technology
Prof. dr. J. Westerholm, Åbo Akademi University, Finland

This work was financially supported by the Global Geo Health Data Center (Utrecht University, the Netherlands) and the Research IT innovation programme (Utrecht University, the Netherlands).

ISBN 978-90-6266-620-1

DOI 10.33540/1127

Copyright ©2022 Kor de Jong

Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt door middel van druk, fotokopie of op welke andere wijze dan ook zonder voorafgaande schriftelijke toestemming van de uitgevers.

All rights reserved. No part of this publication may be reproduced in any form, by print or photo print, microfilm or any other means, without written permission by the publishers.

CONTENTS

1	INTRODUCTION	1
1.1	Model development	3
1.2	Modelling frameworks	3
1.2.1	Agent-based modelling	6
1.2.2	Field-based modelling	7
1.2.3	Integrated agent- and field-based modelling . .	8
1.2.4	Scalability	9
1.3	Challenges	11
1.3.1	Representing geographical state	11
1.3.2	Scalable computing	13
1.4	Objectives and questions	13
1.4.1	Representing geographical state	14
1.4.2	Scalable computing	15
1.5	Societal and scientific relevance	15
1.6	Outline	16
I	REPRESENTING GEOGRAPHICAL STATE	
2	A CONCEPTUAL DATA MODEL FOR SPATIO-TEMPORAL OBJECTS	21
2.1	Introduction	21
2.2	Requirements of the data model and previous work . .	24
2.2.1	Fields	25
2.2.2	Agents	27
2.2.3	Agents with internal variation	27
2.2.4	Networks between agents	28
2.2.5	Useful in a modelling language	29
2.2.6	Integration of fields and agents in a data model	30
2.3	Conceptual data model	32
2.3.1	Elements of the conceptual data model	32
2.4	Addressing the requirements	36
2.4.1	Representing fields in the conceptual data model	36
2.4.2	Representing agents and agents with internal variation	37
2.4.3	Representing networks between agents	39
2.5	Software implementation and case study	40
2.5.1	Software stack	40
2.5.2	Prototype implementation of key data types . .	42

2.5.3	Case study model	43
2.5.4	Implementation of the case study model	44
2.6	Discussion and conclusion	48
2.7	Appendix	51
2.7.1	General structure of the data model in HDF5 files	51
2.7.2	HDF5 structure for grass and cows phenomena from the cases study	53
3	A PHYSICAL DATA MODEL FOR SPATIO-TEMPORAL OBJECTS	55
3.1	Introduction	55
3.2	Spatio-temporal objects	59
3.3	Storing spatio-temporal objects	62
3.3.1	Arrays of temporal information	62
3.3.2	Spatio-temporal object information	66
3.3.3	Spatio-temporal objects	72
3.4	Implementation	75
3.4.1	Arrays of temporal information	75
3.4.2	Spatio-temporal object information	76
3.4.3	Spatio-temporal objects	76
3.4.4	Python package	77
3.5	Example: deer and biomass model	78
3.5.1	Initialize dataset	79
3.5.2	Write model state	81
3.6	Discussion	82
3.7	Conclusion	83
3.8	Appendix	84
3.8.1	Object tracking	84
3.8.2	Representing time	86
II	SCALABLE COMPUTING	
4	A MODELLING FRAMEWORK BASED ON AMT	89
4.1	Introduction	89
4.2	Model development using map algebra	93
4.3	Asynchronous many-tasks and HPX	95
4.4	Method	100
4.4.1	Implementation	100
4.4.2	Scalability and performance	102
4.4.3	Usability	105
4.5	Results	106
4.5.1	Scalability and performance	106
4.5.2	Usability	110

4.6	Discussion	111
4.6.1	Scalability and performance	111
4.6.2	Usability	112
4.6.3	Future work	114
4.7	Appendix	115
4.7.1	Algorithms	115
4.7.2	Scripts	118
5	FLOW ACCUMULATION ALGORITHMS BASED ON AMT	121
5.1	Introduction	121
5.2	AMT, HPX, and the LUE modelling framework	125
5.3	Flow accumulation	127
5.3.1	Overview	127
5.3.2	Parallelization	130
5.3.3	Application of AMT	130
5.3.4	Algorithm details	132
5.4	Experiments	134
5.4.1	Algorithm	135
5.4.2	Performance and scalability	135
5.4.3	Composability	138
5.5	Results	140
5.5.1	Algorithm	140
5.5.2	Performance and scalability	140
5.5.3	Composability	144
5.6	Discussion	145
5.6.1	Performance and scalability	145
5.6.2	Composability	146
5.6.3	Future work	147
6	SYNTHESIS	149
6.1	Representing geographical state	149
6.1.1	Answers and implications	149
6.1.2	Remaining challenges	154
6.2	Scalable computing	155
6.2.1	Answers and implications	155
6.2.2	Remaining challenges	158
6.3	Integrated agent- and field-based modelling	160
6.3.1	Implications	161
6.3.2	Remaining challenges	162

III APPENDIX	
A SOFTWARE	167
BIBLIOGRAPHY	169
SUMMARY	189
SAMENVATTING	195
ACKNOWLEDGEMENTS	201
CURRICULUM VITAE	203

INTRODUCTION

To better understand how an environmental system works, or to predict the state of a system in the past (hindcasting), in the future (forecasting), or under explicitly modified conditions (scenario analysis), it can be useful to simulate the environmental system and the change thereof through time, using computer models. In many cases, computer simulations are the only feasible way to achieve these, for example when the temporal or spatial extents of the considered system are large, the simulated environment is inaccessible, or because of ethical concerns. The outcomes of some of these models can have an effect on the lives and actions of many people, either directly or because they influence the decisions made by politicians. Models that potentially have this implication for example simulate the effect of human movements on the spread of SARS-CoV-2 [171], simulate the effect of climate change on food insecurity [16], or assess the global water scarcity and groundwater depletion [160].

The focus of this research lies on computer models simulating geographical systems. These are systems that typically have a temporal extent ranging from minutes to geological timescales, and a spatial extent ranging from local to planetary scale. Phenomena in the real world are often represented in computer simulation models in very diverse manners. For example, trees in a forest can be considered as discrete entities, each having a unique location in space. In an agent-based model [68] this would be represented by a collection of individual objects [169]. Other models that use this approach include for example those that simulate transmission of SARS-CoV-2 [149], the interactions between pedestrians and cyclists [2], and the effects of different decisions related to the food market on food security [124]. On the other hand, phenomena exist that are continuous in space, like the surface elevation of an area. In a field-based model [61], this would in general be approximated by a raster data structure. Examples of models in which the simulated systems are represented by continuous fields simulate soil erosion [136], wildfire [181], and the distribution of freshwater availability [160]. To achieve an optimal representation of environmental systems, the two different representations, objects

and fields, often need to be combined. This allows the simulation of interactions between individual agents, between agents and their environment, and of changes in the environment. This has been used, for example, to simulate elk migration in Yellowstone [14], the impact of payments for ecosystem services on human-environment systems [5], the spread of cholera in a refugee camp [38], and the dynamics of trout populations under anthropogenic environmental change [7].

The computer models focused on simulate systems which are—in principle—constantly changing: trees grow continuously, and erosion processes may continuously lower the surface elevation. Forward simulation models numerically approximate these continuous changes, where state s of the system is represented for a certain point in simulated time t , after which this state is incrementally updated for future points in simulated time $(t+1, t+2, \dots)$. An application of such a model is weather prediction, that typically uses the known atmospheric conditions at some point in time as the initial state and predicts subsequent states starting with this initial state and the relevant simulated processes that update this state. A single time increment in a forward simulation model can then be expressed as

$$s_{t+1} = f(s_t) \tag{1.1}$$

in which f is the state transition function with which the processes are represented that project the system's state s from one point in time to the next [95, 178]. The subsequent time increments do not have to be uniform, but they often are.

A key issue that is limiting the development and use of computer models for simulating geographical systems, is related to their performance and scalability [4, 17]. Increasingly, more information relevant for use in computer models are becoming available. From new sources of data, like new generations of satellites and social media, information can be obtained that often has a larger temporal and spatial extent and resolution than previous ones. The availability of this information allows researchers to represent geographical processes in more detail in their models. As a result, models are increasingly becoming larger and take more time to execute, or cannot be executed at all. To solve this issue, the performance and scalability of these models must be improved.

In this thesis, we do not touch upon the issue of how well a model is able to represent a geographical system for a certain purpose. Instead, the focus is on the software with which computer models for

simulating geographical systems can be implemented. In particular, the focus is on the representation of real-world phenomena in models, and the capacity of models to simulate large systems.

1.1 MODEL DEVELOPMENT

Given the focus on the software aspects of model development, in this thesis a model development project is considered to be similar to a software development project that has the specific purpose of developing a model. A model development project then starts with an initial set of functional and non-functional requirements. Functional requirements describe which processes the model should simulate and how, while non-functional requirements describe all other aspects, like the maximum amount of time the model is allowed to take to finish calculating the set of state transitions. An initial design is then created and a first version of the model is implemented. During the implementation phase, and afterwards, for example while calibrating, validating and performing modelling studies, the requirements are often updated, necessitating an update of the model. This process of changing requirements that necessitate an update of the model might go on for as long as the model is relevant and used.

The time it takes to (re)implement a computer model determines for a large part the time it takes to perform the whole model development process. Ideally, model requirements never change and implementing a model takes little time, but there are multiple reasons why this is often not the case (Table 1.1). Modelling frameworks can be used to decrease the time it takes to create a first version of a model and to be able to respond more quickly to changing requirements.

1.2 MODELLING FRAMEWORKS

Modelling frameworks provide the model developer with building blocks for developing models. Because of the higher level of abstraction, implementing a model requires less code and thus takes less time (Figures 1.1 and 1.2). When building a model without using a modelling framework, the model developer has to develop more code, some of which is unrelated to representing the simulated system's state and processes. For example, the model developer also has to implement routines for performing I/O and the modelling operations themselves. In a modelling framework this functionality is already

Functional

State or processes must be represented differently.

State or processes must be removed or added.

Model must output other state.

Model must be applied in different study.

Non-functional

Performance or scalability must be improved.

Model must be ported to a new platform.

Bugs must be fixed.

Design must be changed to accommodate new functional requirements.

Quality of the code must be improved, also to allow others to understand it better.

Table 1.1: Examples of reasons for changing model requirements. Some of these may take many months to address.

available, and because a framework is used by multiple model developers to build multiple models, bugs are detected sooner, and multiple model development projects benefit from their fixes.

In this thesis the focus is on modelling frameworks that can be used to simulate a relatively broad range of processes in the domain of geographical simulation modelling. Software that can be used for this purpose ranges from software libraries like Mesa [102] and PCRaster [44], to integrated development environments like MATLAB [80] and NetLogo [173]. A requirement is that information can be positioned in space, and preferably in time as well, and that a collection of operations is available for manipulating this information, for use by the model developer.

Using a modelling framework requires less detailed knowledge about software aspects not related to representing a system's state and processes. An important advantage therefore is that more domain experts can build models themselves, instead of having to rely on a software developer. This prevents errors introduced while communicating model requirements between the domain expert and the model developer. Other advantages are that it supports the exchange of model components between model developers, and it makes it more feasible for model developers to try out alternative representations for simulating the same system [97].

Representing the state s (Equation 1.1) of a geographical system at a certain moment in time in a simulation model involves mapping real-

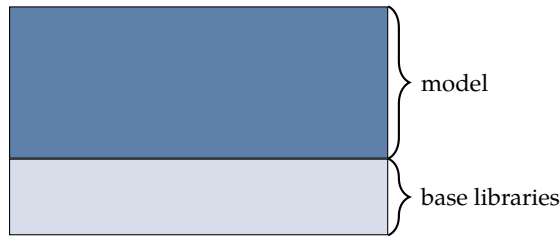


Figure 1.1: A model built from scratch, without using a modelling framework, consists of a relatively large amount of code. The blue part represents the amount of code the model developer needs to write for a model.

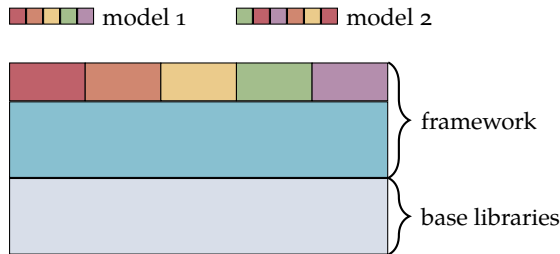


Figure 1.2: Models built using a modelling framework consist of relatively small amounts of code, in which pre-built building blocks are combined. The upper layer of coloured boxes of the framework represents the building blocks. Each string of small coloured boxes represents the amount of code the model developer needs to write for a model.

world natural phenomena to data structures provided by the modelling framework. With these data structures those aspects of the state that are relevant for the simulation must be captured. For example, when simulating the effect of the changing spatial distribution of biomass on the spatial distribution of deer through time, there are various kinds of information that need to be captured: the spatial pattern of a single continuous biomass field through time, the spatial distribution of herds of deer through time, and probably additional properties associated with each deer, like its age and weight.

Similarly, representing the state transition function f (Equation 1.1) in a simulation model involves mapping environmental processes to a set of operations provided by the modelling framework. For example, in a model simulating soil erosion, a flow accumulation operation can be used to partition the amount of rainfall reaching the ground surface into an amount that infiltrates into the soil and an amount that flows downstream as surface runoff.

Modelling frameworks that support forward simulation models in the geosciences and biological domains can be divided into those that are optimized for simulating collections of spatially discrete agents, and those that are optimized for simulating continuous spatial fields of information. These frameworks are called agent-based modelling¹ frameworks and field-based modelling frameworks, respectively.

1.2.1 *Agent-based modelling*

In agent-based modelling, the focus is on representing real-world entities by collections of software agents [68, 148]. Since, in general, the properties and behaviour of each kind of agent is unique, the model developer has to be able to define each of them as part of the model definition. Agent state and behaviour can be conveniently expressed in model code by class definitions, which is the reason why the object-oriented programming paradigm is currently the dominant approach used in the implementation of these models [68]. For example, in Listing 1.1 the behaviour of a herd of deer is simulated by defining a deer class (line 1), including the properties and behaviour of an individual deer, and instantiating it a number of times (line 15). During the execution of the model, the collection of deer instances is iterated over and member functions are called on each of the instances, by a scheduler (line 21). In an agent-based model, most of the state of the simulated system is represented by the properties of all agent objects in the model.

¹ Within the domain of ecology, agent-based modelling is also called individual-based modelling. Here, the more general term is used.

Listing 1.1: Example of part of an agent-based model expressed using Python. The `abm` prefix is used to highlight code that could be provided by a framework.

```

1 class Deer(abm.Agent):
2     def __init__(self, id, model):
3         # Construct deer instance
4         super().__init__(id, model)
5
6     def update(self):
7         # Update instance
8         # ...
9
10 class Model(abm.Model):
11     def __init__(self, nr_deer):
12         # Construct model instance
13         super().__init__()
14         self.scheduler = abm.SomeScheduler()
15         for d in range(nr_deer):
16             # Construct collection of deer
17             self.scheduler.add(Deer(d, self))
18
19     def update(self):
20         # Scheduler calls update on each deer
21         self.scheduler.update()
22
23 model = Model(nr_deer)
24 for t in range(nr_time_steps):
25     model.update()

```

Examples of agent-based modelling frameworks are MASON (Java, [115]), Mesa (Python, [102]), and Repast (Java, Python, C++ among other languages, [126]). NetLogo is an agent-based modelling framework, which does not use an existing general purpose programming language supporting the object-oriented programming paradigm, but provides its own domain-specific language to define models [173].

1.2.2 Field-based modelling

Field-based modelling frameworks describe the modelled system using continuous fields of information about the environment [61]. The model developer defines processes in terms of fields and operations. For the sake of computational efficiency, the raster data model is widely used for representing the fields. Modelling operations are then used to translate these rasters to rasters representing other or future state. To express field-based models, a procedural programming approach

inspired by map algebra [166] is often used, with which modelling operations can be called using function call and operator syntax. For example, in Listing 1.2, framework operations are used to translate an elevation raster into a new raster representing an area's slope (line 3). Note that, in contrast with agent-based modelling, in field-based modelling there is in general no reason for the model developer to be able to define types for model state variables. At each moment in simulated time, the model state is represented by rasters and scalars.

Listing 1.2: Example of part of a field-based model expressed using Python. The `fbm` prefix is used for code that could be provided by a framework. The operators used could also be part of this framework.

```

1 for t in range(nr_time_steps):
2     # Lower the elevation of steep slopes
3     slope = fbm.slope(elevation)
4     elevation = fbm.where(
5         slope > threshold, elevation -= 10, elevation)

```

Examples of field-based modelling frameworks are ArcGIS [46], Google Earth Engine [64], MATLAB [80], PCRaster [44], R [19, 147], TerraME [57], and TGRASS [59].

1.2.3 *Integrated agent- and field-based modelling*

When simulating geographical systems, some state variables are often best represented by discrete objects while others are better represented by continuous fields. In the deer-biomass example, deer can be conveniently represented by discrete objects, and biomass by a field. Given the dichotomy in the approaches of implementing agent-based and field-based models (Section 1.2.1 and Section 1.2.2), expressing processes involving agents using a field-based modelling framework, or expressing processes involving fields using an agent-based modelling framework, is difficult or impossible.

Agent-based modelling frameworks are not designed with field-based modelling in mind. Although continuous layers of information can often be used by agents to sense the environment [126, 158, 173], simulating processes for which the field-based modelling approach is best suited requires functionality, like flow routing operations, that is likely not provided by agent-based modelling frameworks. As a workaround, GIS extensions, like Agent Analyst [85], can sometimes be used to allow their functionality to be used in an agent-based modelling framework.

Implementing an agent-based computer model using a field-based modelling environment can be done for a specific class of agent-based models, in which the location of agents or their properties are represented by a collection of discrete cells. An example of this class of models are cellular automata models, in which each raster cell contains a value, and simple rules are used to update these values, based on the information within a small neighbourhood of cells around each cell. But in a field-based modelling framework it is not possible to express agent-related information at a higher spatial resolution than that of individual cells. Also, expressing agents that overlap in space, or expressing hierarchical relations between agents is difficult.

1.2.4 *Scalability*

A simulation model that has become larger, due to an increase in the amount of data used to represent a system's state or an increase in the number of operations used to represent a system's operations, takes longer to execute when the same amount of hardware is used as before. It is even possible that the model cannot be run at all anymore, because it requires too much memory or too much execution time. To still be able to execute the model as before, it must be able to use additional hardware. Scalability is a property of software with which the ability to use additional hardware efficiently is characterized. When additional hardware is available, a simulation model that scales well takes less time to execute and can execute larger problems.

Models that were not designed to make use of additional hardware, like multiple CPU cores or multiple computers, will only ever use a single CPU core or single computer. Various technologies are available to parallelize and distribute work to multiple CPU cores in multiple computers. For example, OS threads and OpenMP [39] can be used to distribute work to multiple CPU cores within a single OS process, and MPI [123] can be used to distribute work to multiple processes in multiple computers. Most of them leak abstractions of the hardware to the model developer using them, requiring the developer to have knowledge about low level details unrelated to model development. This is one reason why developing parallel and distributed software is much harder than developing serial software.

For a simulation model built using a modelling framework to be scalable, the modelling framework itself must be scalable. This implies that the modelling framework must contain building blocks that, when

combined in models, result in well performing and scalable models, whatever the specific combination of building blocks. This is a very different situation than when implementing a well performing and scalable dedicated model. In that case, all information about the work to be performed by the model is known to the model developer (the code represented by the blue box in Figure 1.1), all of which can be used to implement a well performing and scalable model. The downside of having all this information available is that it increases the complexity of the task for the model implementer. Code from various levels of abstraction, ranging from performing efficient I/O, and distributing work over CPU cores and compute nodes, to the higher level modelling operations used by the model, needs to be developed. There is a risk that this results in a model in which different abstraction levels are mixed, that is difficult to maintain, and of which the performance and scalability are not as good as they could be.

In case of a modelling framework, only the individual building blocks are known (the code represented by the top layer of coloured boxes in Figure 1.2), limiting the options for increasing the performance and scalability of models built with frameworks. This does not, however, imply that models built using a framework will perform and scale less well than models built without using a framework. In contrast to the case of developing a scalable dedicated model, the framework developer can focus on the performance and scalability of the model building blocks and does not have to take the complexity introduced by the actual model into account.

There are multiple options for adding support for scalability to a modelling framework. For example, individual modelling operations can be updated so that they make use of all the hardware available to them. This potentially increases the performance of these operations, because work is assigned to multiple workers, like CPU cores, instead of a single worker. A drawback, though, is that each operation has to wait for the last worker to finish executing the last amount of work before the next operation can start executing. These synchronization points add to the serial fraction of the simulation model, which limits the amount of hardware over which the model can scale well. This approach is used by ArcGIS, for example.

Another option for adding support for scalability to a modeling framework is to let the framework interpret the simulation model and decide how to best compute a result given the available hardware, and then perform the computations. Good results can be obtained by

this approach, as shown by Carabaño et al. [27], but it does add to the responsibilities of the framework. Now at least a language parser, optimizer, and a runtime have to be part of the modelling framework as well.

Finally, (some of) the responsibility of parallelizing and distributing the work to be performed by a model can be forwarded to the model developer. This complicates the model by mixing code related to the representation of the simulated system with code related to how the work should be performed. This is in conflict with the advantage that a modelling framework prevents the model developer from having to know about the runtime details of executing models. When using Repast HPC [34] for example, the model developer is required to implement models using the relatively low-level C++ programming language, and be knowledgeable with using MPI for programming distributed memory systems.

Existing modelling frameworks that provide support for scalable computing are limited by a combination of several factors. Some frameworks only parallelize work within individual operations, some only parallelize work within a single process, some leak details about the parallelization to the model developer, and some are relatively complex to develop.

1.3 CHALLENGES

We have identified two reasons that increasingly limit the usefulness of existing frameworks to model developers. The first reason has to do with limitations in the capability of frameworks to represent and manipulate both discrete agents and continuous fields, which is required to be able to represent a broad range of systems from the geographical domain (Section 1.2.3). The second reason has to do with limitations in the capability of models built with frameworks to scale well over additional hardware, which is required to be able to simulate increasingly large systems of agents and fields (Section 1.2.4).

1.3.1 *Representing geographical state*

Because of the differences between the agent- and field-based modelling approaches, in general it is difficult or impossible to express a field-based model using an agent-based modelling framework and vice-versa. It is therefore impossible to—conveniently—express a sys-

tem in which geographical processes interact with each other using current modelling frameworks. As a solution, models simulating different environmental processes are sometimes coupled, for example using a model coupling framework like the Open Modeling Interface (OpenMI) [66]. OpenMI supports a tight coupling between models, which allows them to exchange information about the various state variables each of them update at each time step. This can be particularly useful in situations in which models are not updated anymore.

But there is merit in a modelling framework with which a model developer can conveniently express a model simulating, for example, the relation between crop growth and soil erosion, where individual plants are simulated using individual agents, and soil erosion using fields. Such a framework allows the developer to use a single modelling paradigm for simulating a broad range of geographical systems. Also, there is then no need anymore to couple multiple models, preventing the additional work needed for that, and the negative implications coupling potentially has on performance and scalability.

However, whether building blocks and a model development interface can be designed that allows model developers to conveniently express integrated agent- and field-based models remains unclear. The approach taken in this thesis generalizes the approach taken by most field-based modelling frameworks. As described in Section 1.2.2, these frameworks provide a set of modelling operations, overloaded for rasters and scalars. In an expression like `elevation + erosion`, `elevation` will likely be a raster, but `erosion` might be both a raster or a scalar. If it is a scalar, the result of the statement is the same as in the case it is a raster with the same value in each cell. The scalar value is a convenient way to encode a spatial field which has a constant value everywhere. Not having explicit information about the spatial distribution of a property value implies this property has the same value everywhere. Both rasters and scalars can therefore be used to represent a spatial field.

The general idea is that when state variables can be considered to be of a similar kind, the actual differences between them can be hidden from the model developer, whenever that is appropriate. This prevents the number of different cases the model developer has to consider. The previous expression does not need to be updated when `erosion` is changed from a scalar to a raster, or vice-versa. This same idea is used here to the integration of agent-based and field-based modelling, in order to find out whether these kinds of modelling approaches can be

generalized. The challenge is to allow the model developer to express a model in terms of state variables and operations that refer to what these variables and operations represent in the real-world, rather than how they are represented in terms of data structures and algorithms. Thus, the first challenge we focus on in this thesis is to bridge the gap between agent- and field-based modelling frameworks. Objectives and research questions related to this challenge are described in Section 1.4.1.

1.3.2 Scalable computing

The second challenge considered in this thesis is to increase the capability of modelling frameworks to execute large models, both in terms of the size of the environmental state s and the size of the state transition function f (Equation 1.1), while still being convenient enough to be used by domain experts without experience in low-level and high-performance computing.

A major complicating factor that arises when scalable software is designed are the synchronization points, which are the locations in the code where all workers—like CPU cores—that are performing part of a larger task have to finish before other tasks can be performed. In the case of load imbalance between the workers, synchronization points often result in some workers having nothing to do. Load imbalance is very common in large environmental simulation models. Instead of using a combination of the two popular approaches for expressing distributed and parallel work, namely communicating sequential processes used by the MPI execution model [123] and fork-join used by the OpenMP execution model [39], the ParalleX execution model [90] will be used, which uses asynchronous many-tasks (AMT). Advantages of this approach we seek are that AMT results in fewer synchronization points, and potentially allows work from multiple operations to be scheduled for execution in parallel, mitigating (some of) the load imbalance. Objectives and research questions related to this challenge are described in Section 1.4.2.

1.4 OBJECTIVES AND QUESTIONS

The overarching goal the research presented in this thesis aims to contribute to is to gain knowledge required to be able to develop a modelling framework for building large models in which both discrete agents and continuous fields can be simulated. This modelling

framework must generalize existing agent- and field-based modelling frameworks for forward spatial simulation models, and be useful in as many research domains as possible. Besides being convenient to use by model developers, models developed with it must perform and scale well.

For this, two objectives and associated research questions were defined. The first objective is related to bridging the gap between the agent- and field-based modelling approaches to allow for the simulation of heterogeneous geographical state variables, and the second objective is to increase the scalability of models created with a modelling framework.

1.4.1 *Representing geographical state*

The hypothesis is that once all model state variables are represented in a similar way, using a single data model, a single set of operations can be defined that can manipulate this state. The objective therefore, is to merge the different data models currently used to represent the state in agent- and field-based modelling frameworks, as a first step towards a single language to express both agent- and field-based models. The associated research questions are:

Research question 1A

How can the different data models currently used to represent spatio-temporal agents and fields be generalized into a single conceptual data model supporting the integrated simulation of both agents and fields?

Research question 1B

How can information related to large collections of spatio-temporal objects and fields be stored using a single physical data model?

A single conceptual data model is relevant for the model developer. It defines how all model state variables are organized. Besides being able to represent all relevant model state, an additional goal for the conceptual data model therefore is that, to be convenient to use in models, it should be as simple as possible. The physical data model is

relevant for modelling framework developers. It allows model state organized according to the conceptual data model to be stored in a dataset for reuse. The model developer should not have to know about the details of the physical data model.

1.4.2 *Scalable computing*

Ideally, it should be possible to develop models using a modelling framework without having to know about how to write scalable parallel and distributed software, allowing more domain experts to develop their own models. The objective is to assess to what extent AMT is useful to reach this goal and the associated research questions are:

Research question 2A

How can the AMT approach be applied in the implementation of a modelling framework which results in models that perform well and scale well over CPU cores and cluster nodes?

Research question 2B

How can the AMT approach be applied in the implementation of a set of parallel and distributed flow accumulation operations that perform well and scale well over CPU cores and cluster nodes?

The answers to these questions can be used by modelling framework developers to help decide whether or not AMT should be used in the implementation, and how.

1.5 SOCIETAL AND SCIENTIFIC RELEVANCE

It is easy to underestimate the societal and scientific relevance of simulation models in general. They help increase our understanding of how geographical processes work, and subsequently they can help make decisions about what future measures to take. For example, using models to find ways to effectively limit the negative effects of high rainfall intensities in urban areas, can save lives and money. It is also easy to overestimate the ability of models to completely represent the processes they simulate. Although the accuracy of weather predictions

for the short term (several days) has increased during the last decades, longer term daily weather predictions are still of limited value.

The societal and scientific relevance of modelling frameworks is that they allow models to be developed in less time, because the building blocks of the models are already implemented and tested. They “only” need to be combined to represent the geographical processes. This requires less time than when models are created from scratch. Also, because of the higher level of abstraction, models based on modelling frameworks are easier to understand, because many details not related to representing system state and processes are hidden. This facilitates open science.

The research presented in this thesis aims to increase the understanding of several aspects related to modelling frameworks for simulating large spatio-temporal systems of objects and fields. All improvements made to modelling frameworks amplify the benefits just described of models and modelling frameworks. Additionally, improvements to the performance and scalability of models created by the future frameworks will make it possible to execute models faster, use larger amounts of more detailed data about the state of the environment, simulate geographical processes in greater detail, and perform more advanced modelling studies, some of which require models to be executed hundreds of times or more.

Whether or not electrical energy be saved by the future frameworks is hard to predict. Although a model that finishes sooner requires less energy to run, the rebound effect may be that it will free up hardware for additional model runs.

1.6 OUTLINE

This thesis is divided in two parts. In the first part, the focus is on the integration of objects and fields. The first step towards an integration of agent- and field-based modelling frameworks is the development of a conceptual data model. Chapter 2 describes how information that is traditionally represented by discrete agents and continuous fields, can be organized in a single uniform data model. This conceptual data model can serve as a blueprint for types used by model developers to represent model state. Given the conceptual data model, Chapter 3 shows how actual model state can be uniformly and efficiently be represented by a physical data model. The physical data model allows storing model state in a dataset for reuse.

In the second part of the thesis the focus is on the scalability of models created with a field-based modelling framework containing modelling operations inspired by map algebra. In Chapter 4 we use the AMT approach to define a type for representing continuous fields and a set of generic modelling operations, and assess the scalability of the operations individually and in combination. Additionally, we look at the usability of the AMT approach in the context of this field-based modelling framework. Given the knowledge gained the AMT-based modelling framework is extended with an additional set of modelling operations operating on a flow direction field (Chapter 5). We again look at the scalability of the operations individually and in combination. Moreover, their composability is considered, which is defined as a property of modelling operations to perform better in combination than when executed individually, one after the other. Composability is a new measure of how well operations are able to mitigate each other's load imbalance.

The thesis is concluded with a synthesis (Chapter 6), in which answers are provided to the research questions defined in Section 1.4 and remaining challenges are highlighted. Additionally, we describe potential implications our results have, and provide an overview of remaining research challenges that need to be solved to gain all knowledge required for developing a modelling framework for simulating large spatio-temporal systems of agents and fields.

Part I

REPRESENTING GEOGRAPHICAL STATE

A CONCEPTUAL DATA MODEL FOR SPATIO-TEMPORAL OBJECTS

Adapted from: M. P. de Bakker, K. de Jong, O. Schmitz, and D. Karssen-berg. “Design and demonstration of a data model to integrate agent-based and field-based modelling.” In: *Environmental Modelling & Software* 89 (2017), pp. 172–189. DOI: 10.1016/j.envsoft.2016.11.016

Dynamic environmental modelling of spatio-temporal systems often requires the representation of both fields and agents. Fields are continuous with values in the whole spatio-temporal domain of a model, while agents are bounded in space and often mobile. It is currently difficult for environmental modellers with limited software engineering background to construct such field-agent models, as modelling frameworks mostly do not support the integration of fields and agents. To overcome this issue, we describe a data model combining fields and agents in a single concept. This data model represents fields, agents and relations by grouping items sharing properties into a phenomenon. The concepts domain, property-set and value handle spatio-temporal attribute representations. The data model is implemented in a software prototype that shows how data on fields and agents is stored and manipulated.

2.1 INTRODUCTION

Dynamic environmental models represent the temporal evolution of the environment by applying a state transition function on the system’s state variables, mostly at each time step or for a series of events occurring over time. Their state variables often vary in the spatial domain as well as in the temporal domain, which requires that an appropriate representation of modelled phenomena has to be used that is capable of storing spatio-temporal information. Two approaches currently exist [49, 60, 137], as shown in Figure 2.1. In the field-based approach, attributes are assumed to have a value defined everywhere in space-time, often discretized in cells or voxels and time steps. Examples of

field-based approaches are aboveground biomass represented by a raster map, where each grid cell value represents the biomass in the grid cell, for each time step, or atmospheric pressure, where each voxel is assigned a pressure value. In the agent-based approach, attributes are assigned to, sometimes mobile, entities bounded in space-time. Agents are sometimes referred to as individuals in ecology and objects in geographical information science (e.g. [62, 68]). An example of the agent-based approach is where the modeller chooses to represent biomass as a biomass value assigned to each individual tree, in order to simulate, for instance, tree growth of individual trees (e.g. [169]).

In a large number of environmental problems, in particular those that require an integrated approach, there is a strong need to use environmental models that combine the agent-based and field-based approaches. This is because many systems contain both phenomena that are better represented by fields and those that are better represented by agents. In addition, there is often interaction between fields and agents (Figure 2.1). In ecosystem models, for instance, animals are often best represented as agents, while their habitat is better represented as a field (e.g. [14, 152]). Other examples include interactions between water users (agents) and groundwater dynamics (field) [28], the influence of weather (fields) on trees (agents) [151], and modelling of air pollution (field) and personal exposure to this air pollution of mobile individuals [150].

The integration of fields and agents in models is however still complicated from the perspective of model implementation, as standardized software is lacking. Generally speaking, field-agent integration can be approached in two ways. One is to build models from scratch with a general-purpose language (Figure 2.2a), for instance Fortran or C++. With such a language, the modeller has to implement software using elementary constructs which requires extensive software engineering skills, especially if the model implementation is challenging, e.g. when the model needs to be executed on a supercomputer. With a few exceptions in certain research areas this exceeds the programming capabilities of most domain specialists. Using these languages comes with a number of disadvantages [45, 97] and shifts the modeller's focus from describing domain processes to implementing numerical algorithms. A solution to this problem is offering model builders specialized programming interfaces tailored to their domain of knowledge (Figure 2.2b). Domain experts then program models using a model building software framework (e.g. [101]), which provides a

programming interface matching the conceptual level of thinking of domain specialists. The building blocks of models can be offered in a generic scripting language or, in its most sophisticated form, as a Domain-Specific Language (DSL) [42, 119], which is a language that matches the semantics of a specific domain of use, improving productivity, verification and optimization [50, 164]. This approach of using model building software frameworks is promising because it becomes easier for domain specialists to program models [79]. However, the current key limitation of model building software frameworks is that they focus on either field-based (e.g. [101]) or agent-based modelling (e.g. [173]), but not both [57]. This is inconvenient because coupling involves additional technical overhead and thus makes it more difficult for domain experts to build models.

The disadvantages of the current situation would be resolved by a model building framework supporting both fields and agents (Figure 2.2c), which preferably includes a domain-specific language capable of manipulating fields and agents. The design of such a language is mostly based on a semantic model of the application domain (e.g. [51]), e.g. a (conceptual) data model. In this chapter, we aim at the development and software implementation of a conceptual and physical data model integrating fields and agents that is useful in a domain-specific language by supporting multi-paradigm operations. The development of a new data model is required as existing data models, with respect to field-agent integration in dynamic environmental models, are still lacking regarding a number of aspects. Recent important contributions have approached integration of fields and agents from a theoretical point of view (e.g. [54, 62, 104]), but the use of these concepts in a framework for dynamic environmental modelling is thus far underdeveloped. Also, existing physical, implemented, data models used in environmental model building frameworks (e.g. [101, 126]) do not integrate fields and agents. Further, to our knowledge, concepts and generic software frameworks that explicitly deal with spatio-temporal attribute variation within the extent of agents do currently not exist. This is relevant in models that model the evolution of individual features in a landscape, including their internal variation, for instance ecosystem models simulating temporal changes in the heterogeneity of foliage in individual tree crowns [169].

A key requirement of the conceptual data model is that it needs to function as an input argument of functions in a domain-specific language for dynamic environmental modelling. We build upon the

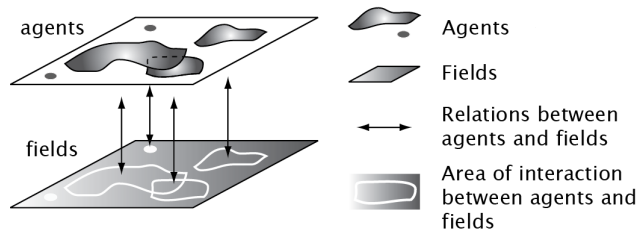


Figure 2.1: An integrated agent-based and field-based model consists of agents, which are discrete entities in space (points, lines, areas) and are possibly mobile, and fields. Fields represent continuous attributes. Many agents have continuous variation within their spatio-temporal extent. Agents and fields interact depending on location and extent of agents.

concept of map algebra, introduced by Tomlin [165] and since then used in many model building software packages for manipulation of fields (e.g. [42, 46]). Map algebra offers syntactically simple statements in the style of algebraic expressions on rasters instead of numbers. In our envisioned modelling language, the input arguments, which are raster maps in the case of map algebra, could also be agents. This implies that at the highest level, fields and agents need to have a common representation in our conceptual data model.

The key question that needs to be answered is how agents and fields can be conceptualized such that these can be represented by a single conceptual data model that can function as input argument in a map-algebra like environmental modelling language. In the following section we therefore specify requirements of the conceptual data model and review existing conceptual and physical data models. Section 2.3 describes the conceptual data model and in Section 2.4 we describe how we address the posed requirements. We also provide a prototype software implementation of the data model based on HDF5, using an example of a vegetation grazing model that integrates field-based and agent-based modelling (Section 2.5). Section 2.6 discusses the approach and results of this chapter.

2.2 REQUIREMENTS OF THE DATA MODEL AND PREVIOUS WORK

The conceptual data model that is being developed is required to represent fields, agents (including agents with internal variation) and relations between these, i.e. networks. Also, it needs to integrate these types and it needs to be applicable in a map algebra like-language.

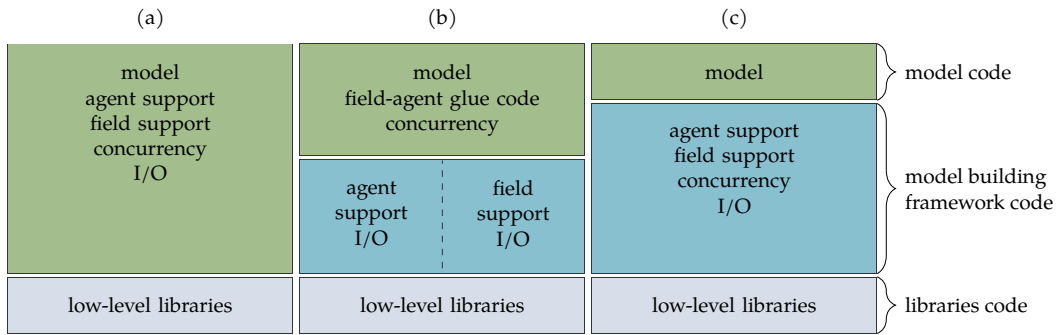


Figure 2.2: Share of model code, framework code and general-purpose programming libraries code in three different approaches to programming models. (a) Using generic, general-purpose programming libraries to build a model from scratch. (b) Using separate model building frameworks for agents and fields. (c) Proposed approach using a model building framework that supports fields and agents. Of the approaches presented in the figure, approach (c) requires the least programming effort from the modeller.

This section discusses these requirements and existing concepts that we can build upon in the development of the conceptual data model. Table 2.1 provides an overview.

2.2.1 Fields

Field-based modelling is defined as approaches that give a spatially and temporally continuous representation of attributes in the environment, which implies that values of an attribute are in principle modelled everywhere in 2D or 3D space and time [55, 101, 142]. In modelling practice, a field is typically bounded in space and time, for example by the study area. Examples of attributes represented as fields are air temperature in weather forecast models [121] and topographical elevation in hydrological models [159] because these phenomena have a value defined everywhere in space and time. When modelling continuous fields, the modeller translates the continuous attribute into a physical data model by discretization, in order to apply numerical algorithms. Widely used spatial discretizations are rasters and TINs [104, 174]. Several programming environments for field-based modelling exist, among which are MATLAB [116], a general purpose modelling environment, and PCRaster [43, 101] for spatio-temporal modelling.

	Conceptual model	Software	Applications
Fields	[24, 114]	MATLAB [116], PCRaster [43, 101], RasDaMan [12]	Hydrological models [159], air pollution modelling [78]
Agents	[68]	GAMA [67], RePast [126], TerraME [57], NetLogo [173], Mason/GeoMason [158]	Economic modelling [139], social simulation [180]
Agents with internal variation	[33] (delatations)	no generic integrated software implementations known	Cloud cover and solar radiation [106], reflection of glaciers [120], air pollution [84], sediment modelling [100], tree growth [169]
Networks	[56, 63]	graph-tool [141], R [147], NetworkX [72]	Farm/household interactions [15]
Integration of fields and agents	[25, 36, 48, 62, 104, 113, 122, 133, 170]	TerraME [57]	Air pollution [84], animals and environment [152], wind and trees [14, 151]
Map algebra language	[52, 165]	Multidimensional Map Algebra [117], MapScript [144]	Vegetation change using Multidimensional Map Algebra [118]

Table 2.1: Overview of conceptual models, software and applications that deal with the specific requirements put forward in the section. The literature that is mentioned shows the emphasis of the offered approach. The columns give a non-exhaustive list of examples.

2.2.2 *Agents*

Next to field-based modelling, modellers can choose to model their system using discrete entities, or objects. The main characteristic of objects as opposed to fields is that they are bounded in space. The spatial characteristic also translates to the temporal dimension, which implies that an object exists for a certain period of time [142]. Table 2.1 provides examples of agent-based models.

From a modelling perspective, objects are often referred to differently. In ecology, the term individual-based modelling is used, while in the Geographical Information Science (GIS) community one often prefers the term object-based modelling. Object-based modelling in GIS has traditionally concentrated on spatial modelling only and little attention has been paid to the time domain [21, 145]. This is opposite to individual-based models in ecology which are often dynamic, and are often called agent-based if applied to societal problems [18]. In these models, individuals or agents are usually mobile and interact with each other and the environment (e.g. [37]). Hence, we choose to use the concept of agent instead of object to represent all kinds of discrete individuals that are bounded in space since it also includes mobility and interaction, which are both important in many dynamic models. This deviates from the traditional use of object as opposed to field, which we view as having a subset of the properties of an agent.

Many specialized agent-based libraries, frameworks and tools exist [1, 37]. Some are not intended to explicitly model spatial behaviour (e.g. [3]), but many allow for explicit spatial modelling, of which GAMA [67], GeoMason [158], NetLogo [173] and RePast [126] are notable examples. Field-based modelling in these agent-based frameworks is often troublesome. Except for GAMA [67] and NetLogo [173], agent-based modelling frameworks usually require modellers to use general-purpose object-oriented data models and languages [37].

2.2.3 *Agents with internal variation*

Many research questions include agents that have a continuous spatial and temporal variation within their spatio-temporal extent. For instance, in a number of air pollution cloud models [84], the movement of clouds is simulated, including spatio-temporal variation of pollutant density within a cloud. Likewise, in vegetation models simulating competition for solar radiation between individual trees [169],

the biomass of a tree crown may vary within its spatio-temporal extent. Other examples include channel belts [100], spatial variation of albedo over glaciers [120] or solar energy models simulating heterogeneous cloud covers and their effect on solar radiation diffusion (e.g. [106]). These examples reveal that systems that can be represented as agents often also have properties that vary continuously within the spatio-temporal extent associated with each agent.

Few concepts are designed that model agents with internal variation and to our knowledge there is no conceptual data model that represents such agents, although the Deltatrons introduced by Clarke [33] are an important contribution and starting point. The same goes for software implementations. To our knowledge no framework supports the modelling of spatio-temporal bounded agents with internal variation. As conceptual models and software for agents with internal variation are lacking, models using this representation are currently programmed from scratch using general-purpose programming languages or programmed by adding components to field-based or agent-based software frameworks (e.g. [84, 100]).

2.2.4 *Networks between agents*

Agents often are part of a network, e.g. social and physical networks (e.g. [15, 20, 29, 138, 151]). Networks can have temporal variation that is unrelated to the variation in the attributes of the connected agents. For instance, the duration and properties of an interaction event between persons is often different from other temporal attributes of the persons.

Peuquet [142] and Kuhn [108] argue that a data model for discrete entities ideally incorporates the representation of networks, whether they are spatially explicit, such as roads and rivers, or implicit, e.g. social networks. Network and graph theory in general is well established, and also in the area of spatial modelling literature deals with networks (e.g. [56, 63]).

Specialized software exists to model networks, often as libraries designed for particular programming languages, e.g. NetworkX [72] and graph-tool [141]. Besides these, other frameworks, such as R [147] and Repast [126] have network programming capabilities.

2.2.5 *Useful in a modelling language*

In dynamic environmental models, fields and agents often need to be connected (e.g. [14, 49, 151, 152]). Figure 2.1 schematically shows the ideal situation in which fields and agents (including agents with variation within their extent) are represented and dynamically interact within a system. Consequently, a modelling language is required that facilitates the integrated modelling of systems containing fields and agents.

We envision that map algebra [166] is a suitable principle as foundation for such a modelling language, because it provides modellers with a technique to manipulate data at a high conceptual level, corresponding to the way modellers think of spatial data. Tomlin [166] developed map algebra as an algebra to manipulate raster maps, i.e. fields. Currently, map algebra is widely used for manipulating raster maps using algebraic notation. In Esri ArcGIS [46], map algebra is implemented as a calculator in which raster maps can be manipulated and combined. The map algebra concept is also realized in MapScript [144] and TGRASS [59]. Likewise, PCRaster [43, 101] includes operations on maps that implicitly iterate over cells in the map. Extensions of the original map algebra have been studied [26, 117, 172], but to our knowledge not in the domain of agent-based modelling. In the following, we show in a very simple manner how we aim to suit map algebra for modelling both fields and agents.

Consider the map algebra statement:

```
map.c = map.a + map.b
```

In this statement, where a , b and c are variables, two maps are combined into a new map. This is done by applying the operation $+$ to the values of the variable a and b for each cell in the map using implicit iteration. Field-based modelling frameworks, such as PCRaster [43, 101] have implemented this style of map algebra. The underlying data model contains rasters for the maps. If we consider agent-based modelling frameworks, it becomes clear that their data model and high-level modelling perspective is very different. Suppose that one wants to apply the same addition operation to attributes of agents in a particular simulation. In agent-based tools, such as NetLogo [173], one typically has to program the procedure using explicit iteration over agents:

```

agents = [agent definition and instantiation]
for agent in agents:
    agent.c = agent.a + agent.b

```

This way of programming at a low level of abstraction is not the most convenient for modellers. Firstly, when an object-oriented approach is used in model development, which is usually the case in agent-based modelling [1], one has to handle the process of object definition, instantiation and management. Secondly, in the part of the model where agents are updated to represent their evolution over time, one has to take care of control flow procedures to iterate over agents. Together, this requires a considerable amount of code which possibly results in a time consuming and error prone development process. Thus, we propose to eliminate the low-level procedures in agent-based modelling, by following an approach similar to map algebra:

```
agents.c = agents.a + agents.b
```

Now, the operation iterates implicitly over all agents in `agents`, adding variable `a` to `b` for each agent. Next, we can make the arguments for fields and agents uniform, leading to a field-agent algebra similar to map algebra:

```
phenomenon.c = phenomenon.a + phenomenon.b
```

Now, `phenomenon` can take the place of a field or an agent. To be able to do this, we need a data model that can represent both fields and agents. A common data model instead of two separate data models comes with clear semantics for the modeller, both when modelling fields and agents. This is an advantage when using a domain-specific language. Further, many phenomena appear to have characteristics of both fields and agents, as will be shown below. In a common data model, we can address this more easily compared to a situation where different data models are used to represent fields and agents.

2.2.6 *Integration of fields and agents in a data model*

The preceding section has shown how we envision the integrated modelling of fields and agents. It was argued that a common data model for fields and agents is required to reach this objective.

Much work has been done on the design or improvement of integrated representations from theoretical and implementation oriented perspectives. Liu et al. [113] introduce the General-Field as a generalization of objects and object-fields. General-fields are constructed

by mappings from fields to object-fields to objects. Geo-atoms and geo-dipoles are introduced in Goodchild et al. [62]. Moreno et al. [122] present the VecGCA model, replacing raster-based cellular automata by a set of geometrically evolving objects (polygons) in order to dismiss the sensitivity of cellular automata to scale. Transition rules determine how the objects transform during a simulation, supported by the vectorization and rasterization between rasters and objects. Data models proposing an integration of objects and fields have been described in Câmara et al. [25], Kjenstad [104], and Voudouris [170]. Câmara et al. [25] presents the geographical object, under which simple, composite, homogeneous and non-homogeneous geo-objects are classified. A minimal set of common GIS operations on these types of geographical objects is introduced. The PGOmodel [104] is introduced as a base model for fields and objects in GIS. Kjenstad [104] conceptualizes fields as a set of functions from a parameter space domain to a value space that is multidimensional instead of a mapping from a geometry domain to a value space. It is not explained how the temporal dimension is accounted for in this model. Voudouris [170] proposes an extension of Kjenstad [104] to allow for the modelling of object-fields [36] and uncertainty and semantics. In the Object-Field Model reflective phenomena are linked to `Elementary_geoParticles`, which are central in the model. The `Elementary_geoParticle` itself does not have predefined geometry, but gets its definition from `attributeset`, `behaviourset`, `semantic` and `uncertainty` classes. Ferreira et al. [48] takes a different perspective on spatio-temporal data representation. It is shown by Ferreira et al. [48] that three data types, that is time series, coverages and trajectory, are sufficient to represent changing objects and events. An algebra on these data types allows for the manipulation of spatio-temporal data. However, the model does not represent relations and the presence of different data types still requires specific operations dependent on the data type. Yuan [177] and Li et al. [111] approach integration of discrete and continuous dynamic modelling via the representation of events, processes and states.

The Open Geospatial Consortium (OGC) has worked on standardization for geographic data representation and exchange, among which is the OGC General Feature Model [133]. Several abstract specifications have been designed (e.g. [130–132]) in order to provide programmers with standards for implementations in which data interoperability and exchange are important [175]. The OGC specifications for representing geographical data have not yet resulted in an integrated data

model that is implemented in a modelling language for heterogeneous systems.

As was argued in the introduction, the distinction between the field-based and agent-based approaches can be found in software used in modelling of environmental systems. Geographic Information Systems are still divided into the raster and vector data models. Also, GIS is currently not well suited for dynamic modelling. Although the urge to generalize the modelling of geographic phenomena is recognized, it is considered unlikely that GIS data models for dynamic modelling and simulation will emerge in the near future [60]. Specialized frameworks such as PCRaster [101], Repast [126], R [147] (with the correct modules) and Matlab [116] are currently often used for dynamic modelling, but they do not integrate fields and agents within their data model or language. An exception is Garcia de Senna Carneiro et al. [57], which brings together field-based and agent-based modelling in a single framework. The urge to store data on both objects and fields is also recognised in the database community. Baumann [11] identified that conventional DataBase Management Systems (DBMS) are not suited for the storage of raster data. Research in this area of database systems has resulted in the development of array databases such as RasDaMan [12] and SciDB [22] as an alternative for relational databases. However, databases are not well suited for dynamic modelling and simulation.

2.3 CONCEPTUAL DATA MODEL

In this section we introduce a conceptual data model for the representation of spatio-temporal data as described in the previous section.

2.3.1 *Elements of the conceptual data model*

The conceptual data model consists of six basic concepts, namely *phenomenon*, *item*, *property*, *domain*, *property-set* and *value* (Figure 2.3);

- The phenomenon is the top level concept in the data model. It represents a thing or occurrence in the real world system. A phenomenon contains one or more individuals, called items.
- An item is an individual contained in a phenomenon.
- A property is a particular attribute of a phenomenon.

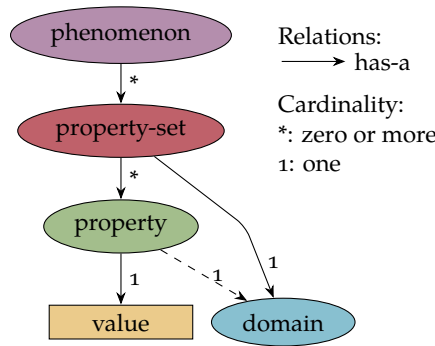


Figure 2.3: Conceptual data model for spatio-temporal data, consisting of five concepts which are connected by *has-a* relations (represented by arrows).

- A domain is the area and time period for which a property exists, for each item.
- A property-set groups properties that have the same domain. The property-set relates to the domain that is shared among the involved properties.
- The value represents the values for each item in a phenomenon for a particular property. Each property has one value.

A phenomenon is related to zero or more properties. Properties represent certain traits of the phenomenon. All domains and all values represent the same number of individuals, i.e. each item has an entry in both the domain and the value.

2.3.1.1 The phenomenon concept

Different from most existing spatio-temporal models in which data is regarded either as consisting of individual objects/agents or as a continuous field, the conceptual data model presented here starts from the idea that everything in the world can be captured by the notion of a phenomenon. This notion is somewhat more specific than for instance the feature concept in ontology, which is used in order to represent a general and homogeneous set of things [71]. The phenomenon concept is used to represent sets of things that share properties, but also share spatial and temporal types.

In the conceptual data model, the concept phenomenon is used to describe a concrete or abstract constituent of the system that is being modelled. For example, if one models an ecosystem in which trees

and groundwater flows are involved, two phenomena could be used to represent the system, trees and groundwater. A phenomenon can represent a single object, for instance the groundwater over the whole study area, or multiple objects, such as individual trees contained in the phenomenon trees. A phenomenon often consists of multiple individuals, which are called items in the data model. This is the case for instance with the example of the phenomenon trees, which includes multiple individual trees as items.

2.3.1.2 *The property and property-set concepts*

A property describes, by means of a reference to a value a particular property, attribute or trait of a phenomenon for a given spatio-temporal domain. A property is included in exactly one property-set and has exactly one value concept (see Section 2.3.1.4).

A phenomenon can have one or more property-sets. A property-set can be thought of as a set of properties sharing a particular spatio-temporal domain, i.e. the area and time period for which the property exists. The property-set refers to exactly one domain and a domain belongs to exactly one property-set. For example, the phenomenon birds has a property-set foraging area, which refers to the spatio-temporal domains of the foraging areas of each of the individual birds in the phenomenon. The property-set groups several properties sharing the spatio-temporal domain. The properties residence time and amount of food collected for example, share the same spatio-temporal domain and hence are grouped into the property-set foraging area. A useful consequence of the binding between properties by means of a property-set is that change in the spatio-temporal traits of a property-set affects all properties the property-set entails.

2.3.1.3 *The domain concept*

The properties of phenomena always have a spatial and temporal location and/or extent. For instance, the maximum height of trees is defined at certain spatial and temporal locations, for example the location of the stem of each tree, while the seed dispersal area of trees is defined for the spatial and temporal extents that can possibly be reached by each tree.

The conceptual data model represents this spatial and temporal definition of a property as the domain. Properties sharing the same

domain are grouped into the same property-set. The domain is the spatial and temporal area or period for which a property is defined. In the domain concept, for each item in the phenomenon (see Section 2.3.1.1), a *domain item* is defined. A domain item is the spatio-temporal location and extent of an individual in the phenomenon. The spatial domain can consist of points, lines, polygons, polyhedrons, but can also contain discretized data, for example cells describing the extent of a phenomenon. Note that in this case the cells do not contain property values but merely dichotomous values describing a phenomenon's presence. The temporal domain has equivalents for the spatial types. A *time point* represents a particular location in time. An *interval* is a *duration* of time with reference to a time point. As can be seen from Figure 2.3, the domain is a necessary concept for the modelling of a property or property-set, i.e. if a property or property-set is defined, there must be a domain defined.

Using the example of trees, two domains could be associated with the phenomenon trees, namely one domain containing a point set of tree locations, and another domain containing a set of polygons representing the tree crowns (see Figure 2.4). For each tree item defined in the phenomenon, a point location is defined in the tree locations domain. The property height for example, is associated with the point domain. The property biomass area is associated with a polygon domain. As can be seen from Figure 2.4 there is no temporal variation in the location of the tree. The temporal domain consists of bounded periods of time in which the change in value of the property is defined.

2.3.1.4 The value concept

Properties of real world phenomena often have meaning because of the associated value within their specific spatio-temporal definition. Such a value can be constant over time and spatial extent, such as the species of a tree, or the weight of a stone within a given temporal domain. The value can also vary continuously in space or time, such as the height of a tree crown or the attenuation of a signal over distance.

In the conceptual data model, the value of a property is represented with reference to the domain of the property-set the property belongs to. Since the value is defined for a particular domain, a property has exactly one reference to a value concept (Figure 2.3). The value allows for the modelling of spatial and temporal variation within the domain.

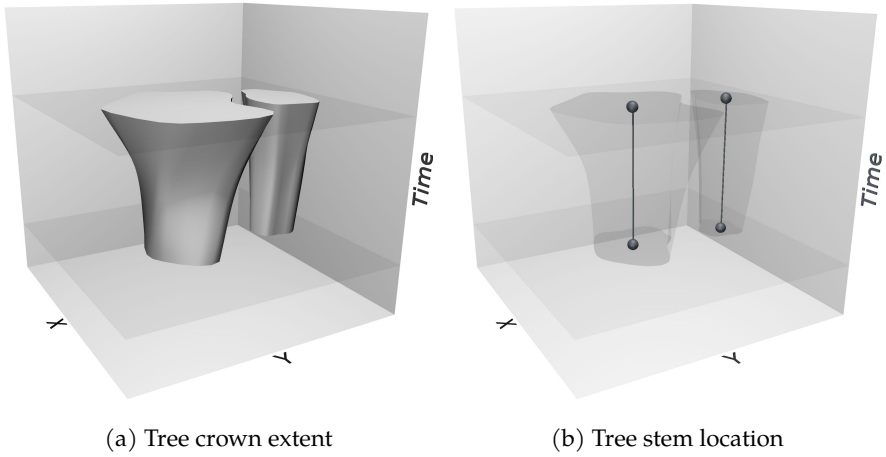


Figure 2.4: Spatio-temporal domain of the trees phenomenon in two spatial dimensions and time. (a) Biomass area domain, extent of tree crowns, growing continuously in time and thus extending in the spatial domain over time. (b) Tree locations domain, point locations of tree stems through time. The tree stems are stationary in space, while the tree extent (shown in (a), and in (b) copied as background) increases.

It does so by supporting the possibility of discretization. If there is spatio-temporal variation for an item, the value discretizes space and time, containing a value for multiple x, y, t . If there is no spatio-temporal variation, the value consists of a single value for each item.

In the example of the biomass of trees, the value will represent the biomass of the crown of a tree individual as shown in Figure 2.5.

2.4 ADDRESSING THE REQUIREMENTS

In this section we explain how the conceptual data model addresses the requirements put forward in Section 2.2 and provide examples supporting our rationale.

2.4.1 Representing fields in the conceptual data model

The conceptual data model is capable of representing classical examples of fields. For instance, a Digital Elevation Model (DEM), storing the level of the area surface as a two-dimensional discretized field of topographical height values over the whole study area, is represented by defining a phenomenon area, which has a property-set surface,

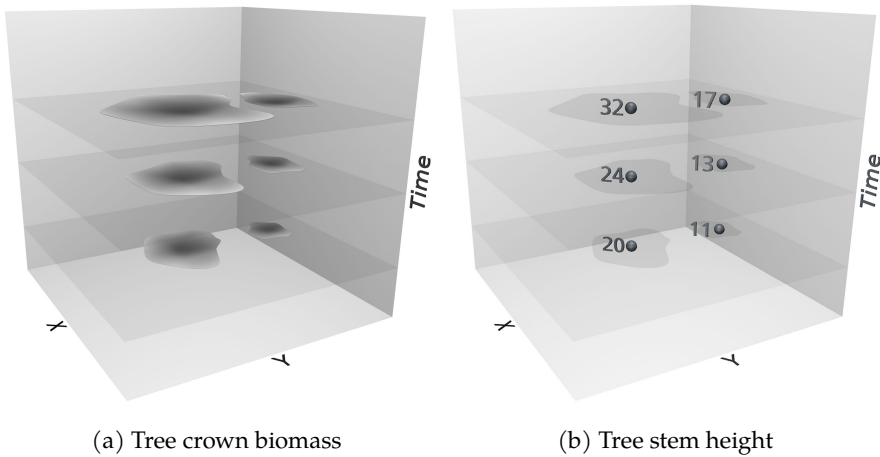


Figure 2.5: Examples of different values. (a) Biomass of the tree crowns represented as a two dimensional array value. (b) Height of tree stems represented as scalar value. These values belong to the tree items represented by the location of the tree stems.

containing the property level (Figure 2.6). In addition, the same property-set can contain other properties that exist in the same domain, such as soil type used here. One domain, extent, is defined and related with the surface property-set, describing the area over which soil data is available. The elevation of the surface is represented by means of the property level. The related value contains the actual DEM in the form of a two dimensional array. The soil type property stores for the same area a field containing the soil types present in the area phenomenon. It contains data for the same area as the DEM and is thus stored in the same property-set surface. The representation works also for three dimensional cases. The property-set subsoil is related with a porosity property, which defines the porosity of the subsoil in three dimensional space, 3D blocks, to be used for instance in a groundwater flow model.

2.4.2 Representing agents and agents with internal variation

Agents are represented using the same layout as fields. For example, Figure 2.7 illustrates how attributes of individual trees represented as agents can be modelled. Different from the case of the field, now the phenomenon consists of several items, all having a domain and a value defined for each property. The tree stem height is a property

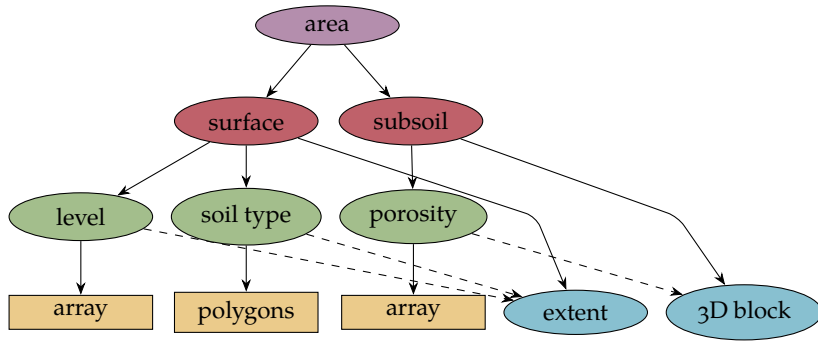


Figure 2.6: A study area with exemplary properties as represented by the conceptual data model. The property-set surface combines two properties sharing a spatio-temporal domain. The property porosity uses a different domain and is therefore put in a different property-set.

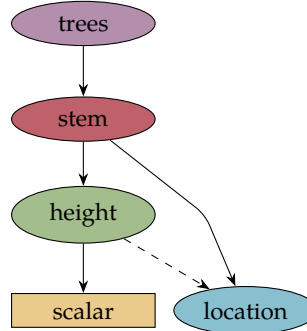


Figure 2.7: Phenomenon trees consists of several individual trees. The point domain and the value representing the stem height define for each tree the property value and the location in space-time.

that is defined for each tree at its location in space and time, given by a single scalar value for each tree.

Agents with internal variation can be represented using a combination of the agent and the field representation. In this case, the value consists of a discretization, allowing a field-based representation for each item in the phenomenon. An example is the crown of a tree agent (Figure 2.8). The biomass may vary within a certain area, i.e. the area covered by the crown. Therefore, the property-set crown is connected with a domain extent representing the crown covered by each tree and the value has for each item a field discretized as a two- or three-dimensional array storing the values of the biomass of the tree inside the tree crown.

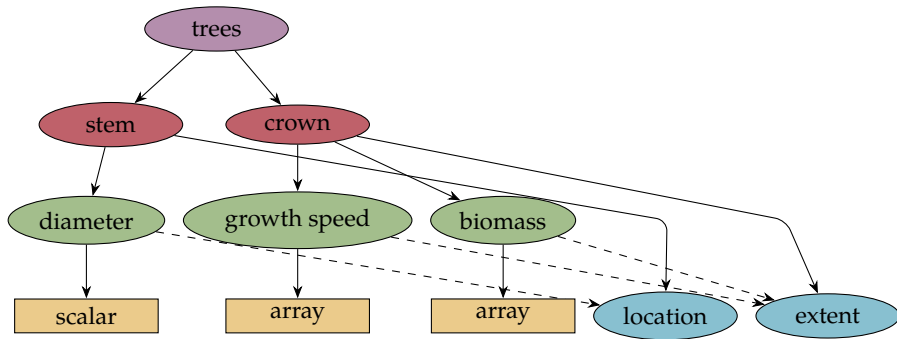


Figure 2.8: Agents with internal variation, i.e. variation within their extent. The combination of a bounded area in the domain with a discretized field in the value represents for each agent the internal variation. This internal variation can exist next to other representations within the phenomenon, as shown by the stem property-set, which is a single location for each tree.

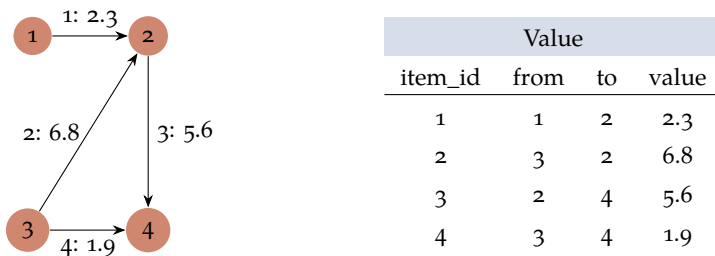


Figure 2.9: Representation of a network. The arrows represent items in the network phenomenon. The dots represent items, for instance persons, in a different phenomenon. The numbers in the dots represent the item ID's. The arrows also have ID's followed by a value, for example the amount of information exchange. The table shows the data contained in the value.

2.4.3 Representing networks between agents

To represent networks, the same concepts from Figure 2.3 are used. This is possible since networks have properties (also called weights) and domains in which they are valid. From the perspective of the modeller it is an advantage that networks can be represented by the same concepts as other phenomena because it means consistency in data storage and modelling language.

Network phenomena associate items of phenomena with each other. This association can occur between items within a phenomenon, or of items between different phenomena. Figure 2.9 schematically illustrates a network. The circles represent for instance items in the

phenomenon people. The arrows represent items, associating people to each other. In network context, these are the nodes and edges respectively. The association is made by means of the properties in the phenomenon, which can be seen as edge weights in a network. The property-set in a relation phenomenon groups weights sharing the same domain of the relation. The value of a property defines for each item (that is, each edge in the network) which items from the associated phenomena (that is, the nodes) are involved and what the value (that is, the weight) of the edge is. Note that the value thus contains references to other phenomenon items.

2.5 SOFTWARE IMPLEMENTATION AND CASE STUDY

This section shows the design and implementation of a physical data model following the concepts of the conceptual data model presented in this chapter. The conceptual data model has been implemented in HDF5 [162], with APIs written in C, C++ and Python. We selected HDF5 for implementation because of its flexibility in defining hierarchical models, portability, interoperability with other programming languages, and support for parallel I/O. The HDF5 framework is currently often used in among other things data intensive research (e.g. [77, 105]).

2.5.1 *Software stack*

The conceptual data model presented in this chapter gives a uniform representation of various kinds of data at a high level of abstraction, where the phenomenon concept is the highest level of abstraction. The phenomenon may contain a considerable range of different types of data. For instance, Figure 2.8 shows the *trees* phenomenon containing both data at point locations and data related to a spatial extent, described by the location and extent domains, respectively. Examples of other data that can be stored in a phenomenon are relations and discrete time series. We call these kinds of data in the conceptual data model data types. Data types are any valid combination of domain and value, such as agent, field and relation.

While different data types are treated equally at the abstraction level of the phenomenon, for performance reasons they each require a specific data structure for physical implementation. Thus, an HDF5 data model has been designed for each data type, resulting in a set of HDF5

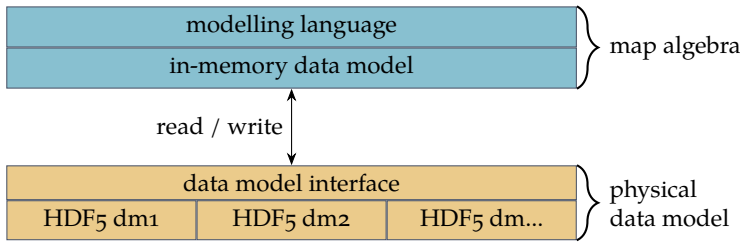


Figure 2.10: Data model implementation. The implementation consists of several HDF5 models (e.g. HDF5 dm1) which are created and maintained by means of the data model interface. Together, these blocks form the physical data model. The map algebra block consists of an in-memory data model reflecting the conceptual data model which reads and writes data from the physical data model. On top of the in-memory data model is the modelling language which is the interface to the modeller.

data models (e.g. for agents and fields) embedded in our physical data model (Figure 2.10). These data models are constructed and managed by a layer on top of HDF5, the data model interface (Figure 2.10), which is written in C and C++. This layer provides an interface between our conceptual data model, which stores data as phenomenon, property-set, property, domain and value (see Figure 2.3), and the various data models in HDF5. Together, the HDF5 data models and the data model interface form the physical data model.

The physical data model is about storage and retrieval of data from disk. It would be inefficient in a modelling context to work directly on the physical data model due to the file access overhead. Hence an in-memory data model (see Figure 2.10) has been built to read and write data from and to the physical data model when needed. The in-memory data model reflects the conceptual data model and allocates the data retrieved from the physical data model in memory. The in-memory data model allows the quick retrieval of data, which is often necessary in calculations.

In addition, the in-memory data model is the basis for a modelling language (Figure 2.10), which has been discussed earlier (Section 2.2) as a requirement of the conceptual data model. The modelling language provides modellers with map algebra like language constructs that manipulate the data in the data model. The modelling language prototype is currently implemented in Python. The behaviour of an operation depends on the operation class and the data type of the arguments. Examples of operation classes are local operations, movement and selection (or querying). In the modelling language, implementa-

tion details are hidden from the modeller, who only needs to think at the level of the conceptual data model in order to understand how to use an operation.

2.5.2 *Prototype implementation of key data types*

We demonstrate the implementation of two data types that enable integrated field-based and agent-based modelling (Figure 2.1), namely one data type for modelling a single, gridded field, and one for modelling a set of mobile agents. The field data type has one or more continuous properties, which are discretized in equally sized cells. The cell values are stored at discrete time steps. The data type consisting of a set of mobile agents stores for each agent a location in space and attributes, which are both updated at discrete time steps. Note that these data types enable only a particular type of integrated field-based and agent-based modelling, as they are restricted to, for instance, fixed time step modelling. Other data types for field-based and agent-based modelling are implemented as well, for instance those representing a set of agents having a spatial extent, or with a domain discretized by irregular time steps. The explanation of the implementation of all data types however requires the design of a formal taxonomy of data types, which is beyond the scope of this chapter. Therefore, we restrict ourselves to the two data types used in a case study.

The two data types are each implemented by a specific HDF5 data model (HDF5 dm1 and HDF5 dm2 in Figure 2.10). These HDF5 implementations make use of the HDF5 objects, which are *groups*, *attributes* and *datasets* [162]. An HDF5 group is the organizing object, and it can contain (multiple) other groups, datasets or attributes. HDF5 datasets objects contain nD arrays of data. HDF5 attributes contain data about HDF5 groups or HDF5 datasets. The structure of HDF5 data sets is analogous to a file system, in which groups are folders and datasets are the files. Using these HDF5 objects, we have designed the HDF5 implementation of the two data types discussed here, the field and the set of agents. We make use of HDF5 groups and HDF5 datasets in order to store the data. Appendix 2.7 discusses the implementation in detail and shows that the HDF5 file structure for both data types is very similar, except for a few details.

A software layer, the data model interface (Figure 2.10), written in C and C++, creates the HDF5 data models, and contains operations to update and retrieve data. The software layer contains templates of

the various data types to allow for the maintenance of the HDF5 data models. A less extensive version written in Python is used in this case study.

2.5.3 Case study model

The case study model represents grass biomass growth and grazing cows. We use a highly simplified representation of the processes retaining key elements of interaction between fields and agents, as our case study is intended only to illustrate the use of the data model and map algebra language.

The change in biomass $B(kg \cdot m^{-2})$ at a location is represented by the logistic growth equation with grazing and dispersion components [99, 127]:

$$\frac{dB_{x,y}}{dt} = rB_{x,y}\left(1 - \frac{B_{x,y}}{k}\right) - g + dD_{x,y} \quad (2.1)$$

where $D_{x,y}$, the dispersion, of a cell (x, y) is determined by

$$D_{x,y} = B_{x,y-1} + B_{x,y+1} + B_{x+1,y} + B_{x-1,y} - n_{x,y}B_{x,y} \quad (2.2)$$

In Equation 2.1, k is the carrying capacity $10 (kg \cdot m^{-2})$, and r is the growth rate $0.02 (day^{-1})$, and g is grazing $(kg \cdot m^{-2} \cdot day^{-1})$. Biomass is represented by a discretized field, using raster cells with a constant size of $1 \cdot m^2$. For each grid cell, Equation 2.1 is solved at a set of time steps with a duration of one day. Biomass removal is represented by $i = 1, 2, \dots, n$ mobile, foraging, cows. Each cow eats biomass at a maximum rate of $g_i = a_i \cdot w_i \cdot kg \cdot day^{-1}$, with a_i , a parameter $0.0008 (day^{-1})$, and w_i , the weight of the cow (kg). The grazing g in Equation 2.1 at a particular grid cell and time step is the sum of the g_i values of all cows i in the grid cell at that time step, divided by the cell area, which is in this case 1. Grazing at a particular grid cell stops when biomass B is less than $0.01 kg \cdot m^{-2}$. The dispersion $dD_{x,y}$ is calculated by summing for each cell the biomass of the directly surrounding cells and subtracting the biomass at the cell itself multiplied by the number of surrounding cells. This number is four for all cells except cells at the border. The parameter d is the dispersion rate.

At each time step the cows move. For simplicity, the movement of the cows is random, based on a bivariate (x, y) normal distribution with the current location of each cow as the mean and a σ of $30 m$. However, the cows only move to a certain cell if the cell contains enough biomass,

that is, if there is at least $0.5 \text{ kg} \cdot \text{m}^{-2}$ available. If this amount is not available, a new random location is calculated until the requirement is met.

The cows increase per time step in weight with an amount of 0.3% of their current bodyweight, until the maximum weight of 750 kg is reached.

2.5.4 Implementation of the case study model

The implementation of the case study model consists of two phenomena, namely grass of the field data type, and cows of the agent data type, shown in Figure 2.11. The grass phenomenon contains one biomass property at the start of the simulation. The cows phenomenon contains a weight property. The model is programmed using the modelling language (Figure 2.10).

The modeller uses a syntax that is related to the concepts in the data model. Following map algebra, the form of the operations in the language is $result = operation(arg_1, ..., arg_n)$, with *result* and arg_i a phenomenon, value or domain.

Listing 2.1 shows the code of how the grazing model is programmed by the modeller. Model parameters are set as Python variables in lines 1–6. Two initial data sets exist, one containing the cows phenomenon and one containing the grass phenomenon. These datasets are loaded using the function `read(HDF5_dataset)`, as is done in lines 7 and 10. Appendix 2.7 provides a description of the file structure of these HDF5 datasets. The variables `grass` and `cows` now have the initial value of the data in the datasets stored in the in-memory data model, i.e. the initial state before the time updates start. Dot notation is used to access these components, in the order *phenomenon.property_set.property*. For example, `grass.area.biomass` gives access to the value of the property `biomass` in the property-set `area` of the phenomenon `grass`. Also, because a property-set refers to exactly one domain, `grass.area` provides directly access to the domain of the property-set `area`.

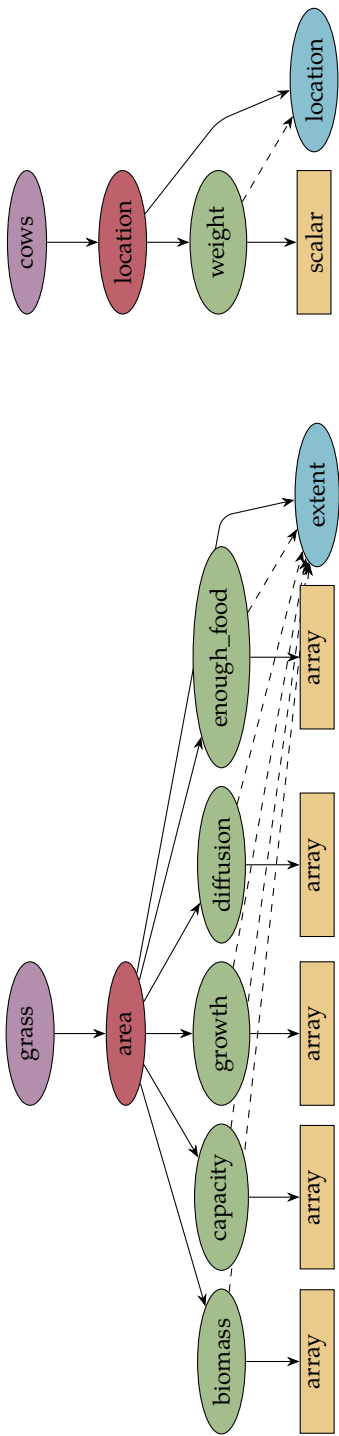


Figure 2.11: Conceptual data model of the grass and the cows phenomenon for HDF5 implementation (see Appendix 2.7) at the end of the simulation of the case study. At the beginning of the model run, in the grass phenomenon only the biomass property exists. In the cows phenomenon, the number of properties does not change.

Listing 2.1: Grazing model programmed in Python using the map algebra modelling language.

```

1 grass_datapath = "../data/grass.hdf5"
2 cows_datapath = "../data/cows.hdf5"
3 growth_rate = 0.02
4 carrying_capacity = 10.0
5 d = 0.01
6
7 grass = read(grass_datapath)
8 grass.area.nr_neighbours = window(
9     grass.area.biomass, "count", "manhattan")
10 cows = read(cows_datapath)
11
12 for t in range(1, nr_of_timesteps):
13     grass.area.capacity = 1 - grass.area.biomass / carrying_capacity
14     grass.area.growth =
15         grass.area.capacity * growth_rate * grass.area.biomass
16     grass.area.biomass = grass.area.biomass + grass.area.growth
17     grass.area.diffusion =
18         d * (window(grass.area.biomass, "sum", "manhattan") -
19             grass.area.nr_neighbours * grass.area.biomass)
20     grass.area.biomass = grass.area.biomass + grass.area.diffusion
21     grass.area.biomass = maximum(
22         0.01, grass.area.biomass - (cows.location.weight * 0.0008))
23     grass.area.enough_food = grass.area.biomass > 0.5
24
25     cows.location.weight = minimum(
26         750, cows.location.weight * 1.003)
27     cows.location = gaussian_move(
28         cows.location, grass.area.enough_food, 30)
29
30     write(grass, grass_datapath, t)
31     write(cows, cows_datapath, t)

```

The model runs for a specified number of time steps with a duration of one day using standard Python iteration (line 12), where the number of time steps is defined outside this script. Lines 13 to 22 represent all terms of Equation 2.1. Note that the operations are applied to all cells on the biomass map. For example, line 13 retrieves the biomass using `grass.area.biomass` and divides the value of all cells by the carrying capacity. The result of the operation is assigned a new property, `capacity`, in the same property-set `area`.

The last term in Equation 2.1, diffusion of biomass, is represented by the *window* operation in line 17, which aggregates over a spatial manhattan neighbourhood (all directly neighbouring cells next to a cell). The inputs of the calculation are the current biomass (calculated on line

16), the diffusion parameter d (line 5) and the number of neighbours for each cell in the `grass.area.biomass`. This last map is calculated in line 8 with a *window(property, aggregation method, neighbourhood)* operation counting for each cell the number of cells at manhattan distance. This means that all cells are assigned four neighbours except cells at the border of the map, which have two or three neighbours. The result of the calculation at line 17, `grass.area.diffusion` is added to the biomass map `grass.area.biomass` (line 20).

In line 21, the cows phenomenon is used to calculate how much the cows eat. The cows eat an amount of biomass proportional to their weight, which is 0.008 percent. This amount, divided by the cell area (which is in this case 1 m^2) is subtracted from the biomass of the cell where each cow resides. Because operations apply to all items in the phenomenon, the operation *cows.location.weight* \times 0.0008 is applied to all cows. Line 21 thus combines the agent and field data types by calculating the new value of the field on the basis of the value of the agent data type. The *maximum(scalar, property)* operation takes for each cell in biomass the maximum of the cell value and 0.01. This represents the model assumption that grazing does not occur below a biomass of $0.01 \text{ kg} \cdot \text{m}^{-2}$.

Line 25 increases the bodyweight of each cow by multiplying the weight property of each cow with 1.003. To avoid that the cows' weight continues to increase, the operation *minimum(scalar, property)* assigns the new weight or 750 to the `cows.location.weight` property, for each cow.

The movement of the cows is represented by the move operation *gaussian_move(agent_property_set, field_property_set, sigma)* (line 27). This operation calculates the new location of each cow by taking its current x and y coordinate and uses these coordinates as the mean in a bivariate (x, y) normal distribution. The standard deviation is provided by the modeller. The move operation generates a new random x and y coordinate on the basis of this distribution. An underlying boolean field, `grass.area.enough_food`, calculated in line 23, determines whether this new location is possible, namely if the biomass exceeds $0.5 \text{ kg} \cdot \text{m}^{-2}$. If not, the *gaussian_move* operation calculates a new location until the location is valid.

The new values that are calculated during the model run are stored in the in-memory data model. We often want to store the (intermediate) model data to disk. Thus, lines 30 and 31 write the content of the in-memory data model as a new time step to the HDF5 files stored at

disk. The new data values become the current state in the in-memory data model at the following time step iteration.

Figure 2.12 shows the state of the model stored to disk at several time steps. Initial biomass is assumed close to zero. It can be seen that the biomass increases while the cows wander randomly over the area. In the beginning, cows are concentrated in a single area. As the simulation continues, the cows search for patches (cells) with enough food. While the simulation proceeds and the biomass increases, the cows seem to form new smaller clusters.

2.6 DISCUSSION AND CONCLUSION

We introduced a new data model for representing several types of data commonly used in environmental modelling. We have shown that this data model is able to combine field and agent data and is suitable for a modelling language that is built upon the concepts of map algebra.

A number of model building frameworks already exist and interfacing between frameworks is in principle possible. However, our conceptual data model can be a valuable contribution to the modelling community. For instance, a number of authors [6, 41, 164] have stressed that it is advantageous to programmers and modellers if the semantics of the language they use is near to their application domain. This is exactly what is provided by our data model: the modeller gets access to fields, agents and relations, and is capable of manipulating this information through a single conceptual representation of the data. Also, it is stressed in the literature that domain-specific languages are important for more efficient and effective model implementation as they take away programming complexity from the user [6, 119]. This is particularly relevant for modellers of spatio-temporal systems because the systems being modelled are becoming increasingly complex and thus require more complicated calculations and coding. Our data model is, as shown here, a relevant step in this direction, as it can be used as argument in a map algebra like domain-specific language. Map algebra is known to be a concept that can easily be grabbed by domain specialists without a strong programming background [117].

The value of the data model is thus to offer an approach for modelling fields and agents. This is not to say that the data model prescribes the way a modeller represents a certain problem. What representation to choose given the research question and the available data is still a task of the modeller. Our conceptual data model does not prescribe

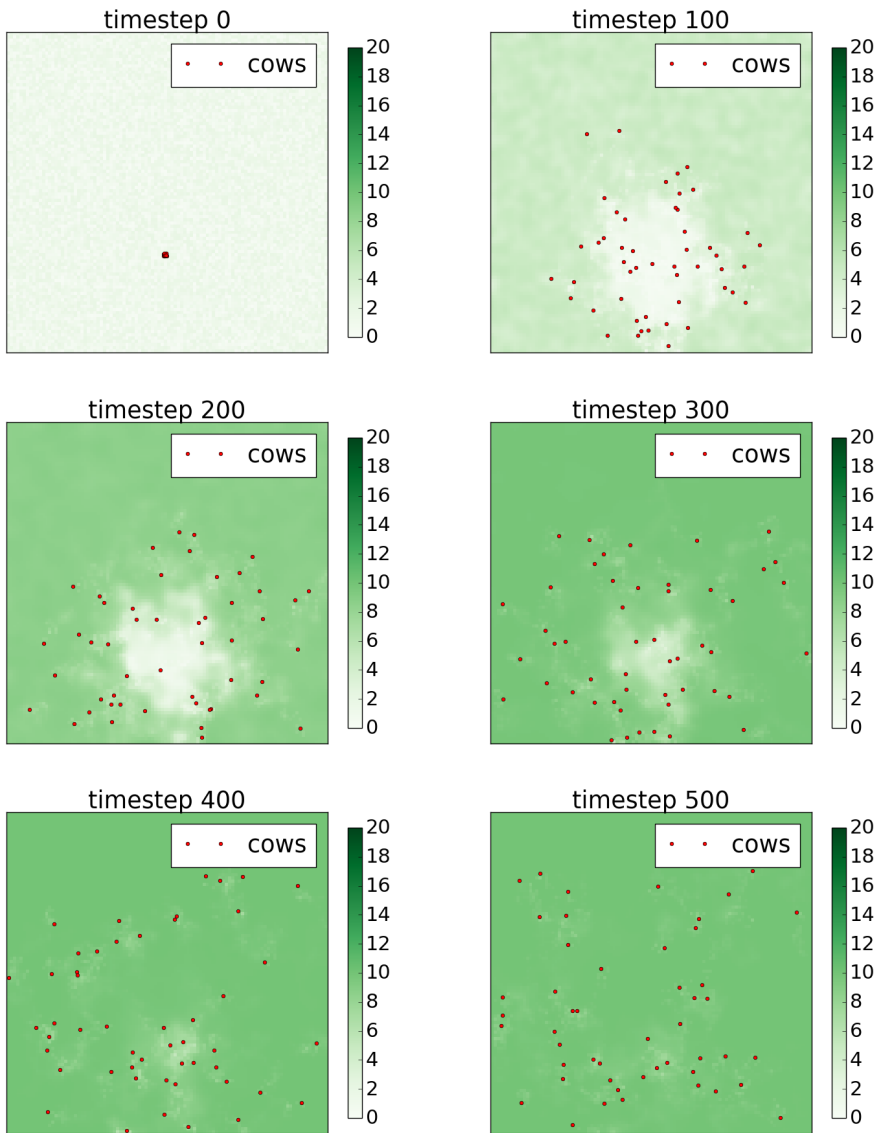


Figure 2.12: grass.area.biomass (shown as a green field) and cows.location (shown as red dots) at 0, 100, 200, 300, 400 and 500 days.

how to represent data in a particular modelling question. For example, in the case of modelling land-use, it is the modeller who decides to represent land-use either as a field, where a nominal attribute value represents the land use type, or alternatively as set of agents, where each individual field is an agent. In a similar fashion, it is up to the modeller to choose the representation of biomass, as a continuous field of biomass values for each pixel, for instance derived from a remote sensing image, or as a set of agents, where each tree is represented by an agent. Our data model can accommodate these different representations, which can coexist, and which can be chosen according to the preferences of the modeller. This flexibility is advantageous because the choice for a representation is usually specific for a particular study, and dependent on, for instance, the research objectives and the data availability.

The idea of integrating the modelling approaches is not new, and we have build upon existing work on conceptual data models for geographic data representation (see Section 2.2). For instance, as is the case for the OGC specification [130], we represent relations separate from the nodes in the relation. Also, like Voudouris [170] we allow the representation of phenomena having properties that are both field-like and object-like, and like Grignard et al. [67] we aim at providing an easy to use modelling language. Still, our conceptual data model is distinguished from existing work in several ways. For example, having multiple individuals with different domains included in a single phenomenon is new. Implementations of agents having internal variation do currently not exist in the agent-based frameworks without the need of ad hoc workarounds (e.g. NetLogo [173], GAMA [67], Mason [158], Repast [126] and TerraME [57]). Also, from the point of view of language design, our work is distinguished from other examples in the literature. For instance, it differs from recent work by Kuhn [109] which aims at developing generic queries for geographic information, but still uses two different representations for objects and fields.

The prototype of the data model and modelling language was demonstrated in Section 2.5. The implementation shows that the conceptual data model can be implemented as a physical data model with a modelling language on top to manipulate the data stored in the data model. The prototype reveals advantages of the work presented in this chapter. Firstly, it shows that it is possible to integrate different data types at a high level of abstraction by implementing several representations

in the lower level physical data model. Secondly, using the modelling language it is possible to manipulate data in the physical data model. An example of this is that operations implicitly iterate over all items in the phenomenon, as is exemplified by the addition of two maps and the *window* operation (see line numbers 16 and 17 in Listing 2.1). Thirdly, the language is capable of performing operations on combined data types, e.g. fields and agents, for example subtracting biomass from the field on the basis of the weight of the cows, and the *move* operation (see line numbers 21 and 27 in Listing 2.1).

Current and future research needs to focus on a number of conceptual and implementation related issues. Ongoing research includes the definition of physical data models in HDF5 for the many other data types that can be derived from the conceptual data model. Two data types (field and agent) were discussed in Section 2.5, which are being extended to more advanced data types such as agents having internal spatial variation. Current research also focuses on supporting high-performance computing of any dynamic environmental model developed with the data model and language introduced here by using the parallel computing version of HDF5. In addition to this current work, a number of important challenges remain for future research. One is that the incorporation of time in the conceptual data model poses questions regarding temporal modelling. Modelling and simulation of continuous time and uncertainty should also be included in our framework. Another open issue regarding the modelling language is the extension of the set of intuitive generic operations that are meaningful given particular combinations of data types. The modelling language (Section 2.5) now consists of a limited number of operations and interactions. As new data types are being developed, the number of combinations of arguments in operations increases, meaning that an approach to assess valid combinations of arguments is needed.

2.7 APPENDIX

In this appendix, the structure of the HDF5 files is explained.

2.7.1 General structure of the data model in HDF5 files

We have designed the HDF5 implementation of the field and the set of agents data types. We make use of HDF5 groups and HDF5 datasets in order to store the data. HDF5 groups represent phenomenon, proper-

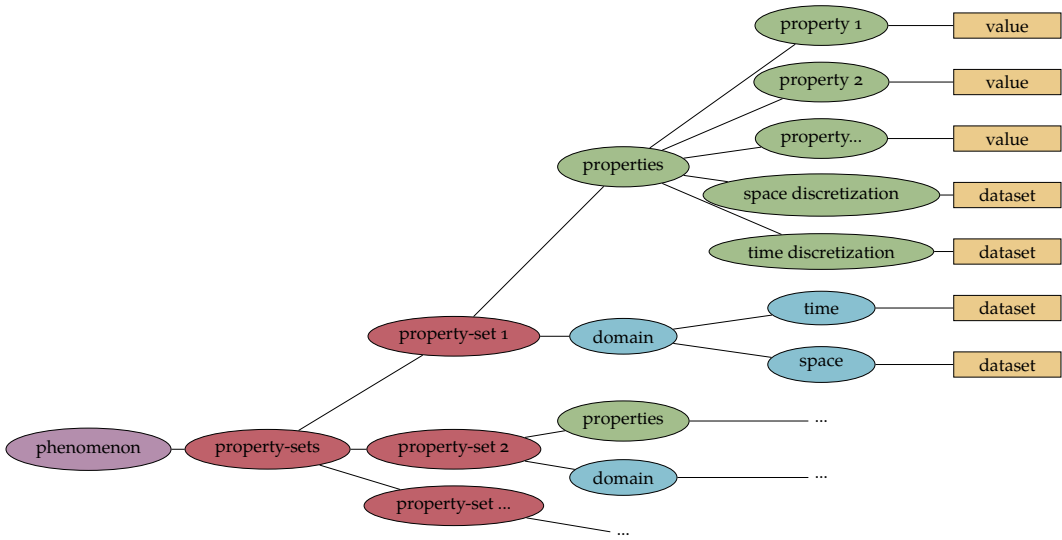


Figure 2.13: Hierarchical structure of HDF5 file of a field data type. Oval shapes represent HDF5 groups and square shapes HDF5 datasets.

ty-set, property and domain. This implies that for each property-set or property a new subfolder exists, which results in a hierarchical folder structure. This hierarchical structure is illustrated in Figure 2.13 and 2.14. HDF5 datasets represent the value concept from the conceptual data model. Furthermore, HDF5 datasets are used to store the data contained in the domain and discretization information. Note that if, for example, a new property is created during the model run, a new HDF5 group is added to the structure.

In Figure 2.13 and 2.14 the concepts of the conceptual data model are recognizable. Some extra datasets are necessary to store additional information on spatial and temporal dimensions of the data (space and time HDF5 groups and related datasets in both figures). Furthermore, for the field data type, we must store how the field is discretized. This is done by means of the space discretization HDF5 group and related dataset. The temporal discretization HDF5 group and related dataset is present for both the agent and the field data type. Because differences between the data types is only made in the domain and value, the general HDF5 structure is almost equal for both the field and the set of agents. The layout of the HDF5 dataset, which is an nD array, however differs somewhat. Since the field data type only consists of one item, this item is represented by one HDF5 dataset, a 2-dimensional array. In the case of the set of agents data type, the HDF5 dataset represents

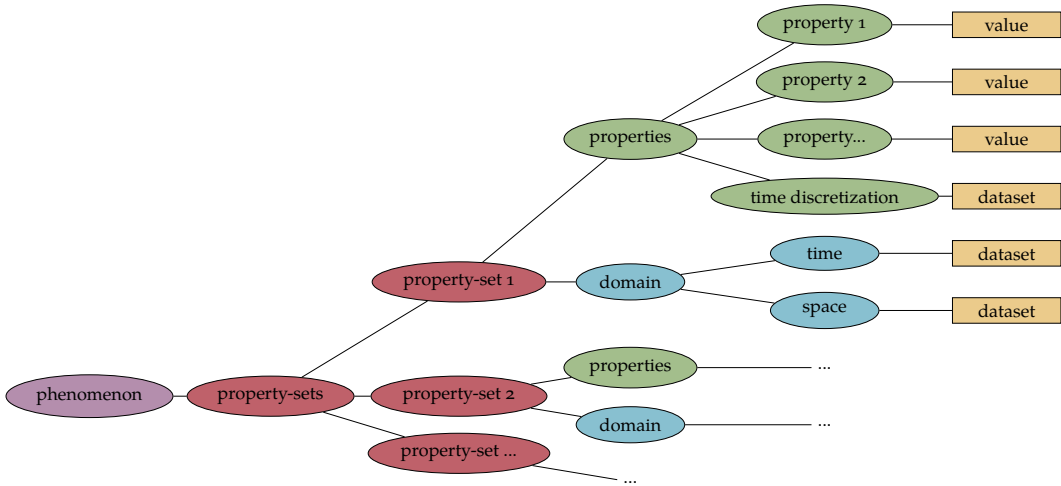


Figure 2.14: Hierarchical structure of HDF5 file of a agent data type. Oval shapes represent HDF5 groups and square shapes HDF5 datasets.

multiple items, that is one, the first dimension of the array represents the item, while the second dimension represent the value.

2.7.2 *HDF5 structure for grass and cows phenomena from the cases study*

The figures show the structure of the files after running the model. Figure 2.15 thus contains all properties that were created while executing the model.

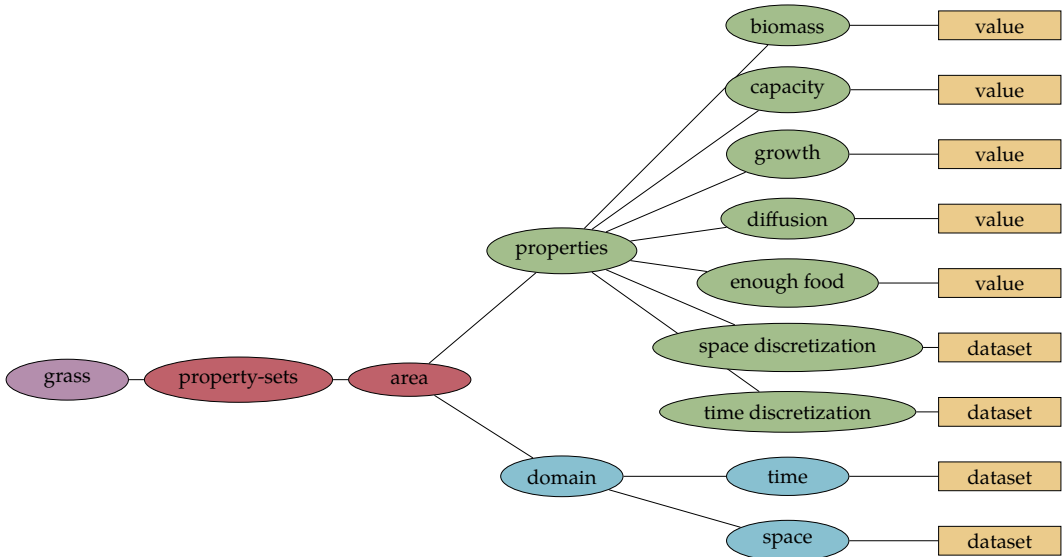


Figure 2.15: The structure of the HDF5 file `grass.hdf5` after running the model (Listing 1 in the main text).

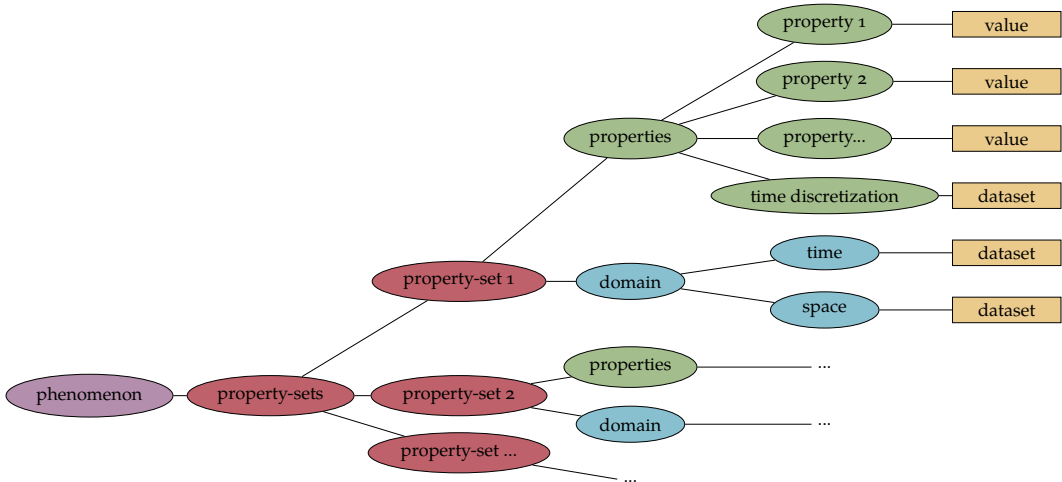


Figure 2.16: The structure of the HDF5 file `cows.hdf5` after running the model (Listing 1 in the main text).

A PHYSICAL DATA MODEL FOR SPATIO-TEMPORAL OBJECTS

Adapted from: K. de Jong and D. Karssenberg. “A physical data model for spatio-temporal objects.” In: *Environmental Modelling & Software* 122 (2019), p. 104553. DOI: 10.1016/j.envsoft.2019.104553

Modellers simulating the state of the physical and biological environment using agents and fields, face the challenge of storing the state variables, like the distribution of biomass and the location and properties of animals. These variables differ from each other, depending on how exactly they represent their state. In this chapter we describe an approach for storing multiple kinds of representations of model state variables using a unified physical data model. We identified a set of aspects in which these representations differ from each other and which have an implication for the data model. Based on these aspects we defined a physical data model for storing spatio-temporal objects, which we implemented in an open source software library. We illustrate how the resulting data model can be used to store multiple kinds of model state variables and explain how this data model can be useful in environmental modelling software.

3.1 INTRODUCTION

When simulating the state of the physical and biological environment using a computer model, model developers face the challenge of storing this state, as represented by the model variables, in one or more datasets for post-processing. Depending on how the model variables represent state, storing each of them may require different physical data models (dataset formats or database designs). For example, when modelling the influence of grazing deer on the spatial distribution of biomass, the state to be stored might be the continuous distribution of the biomass through time and space, the changing location in space of each deer, the changing non-spatial weight of each deer, and the sex

of each deer, which is not dependent on time and space. The modeller might in this case decide to store biomass using a collection of rasters (one raster per simulated location in time), stored in one of the raster formats supported by the Geospatial Data Abstraction Library (GDAL) [58], animal location using a collection of spatial points per animal stored in one of the vector formats supported by GDAL, animal weight by a collection of floating point numbers per animal stored in an SQLite database [156], and sex by a single code per animal also stored in the SQLite database. Since GDAL does not support time, some ad hoc convention is needed in this case to relate information in the datasets to simulated time. An additional convention is needed to relate the information about the simulated deer in the various datasets to each other. Having to use multiple physical data models and conventions is inconvenient for modellers, error-prone and results in models that are less easy to maintain.

Which kinds of representations of state variables are used in environmental models depends on various factors, of which the modelling paradigm used is an important one. Two main paradigms for modelling environmental processes are field-based modelling and agent-based modelling [49, 60, 137]. In field-based modelling, modelled systems are represented by continuous spatial fields of information. These fields are often represented by rasters, like a raster containing amounts of biomass in an area. In agent-based modelling, systems are represented by interacting discrete objects of information that are mostly bounded in time and space. These entities can be representations of physical real-world phenomena, like deer, but they can also represent organizational entities, like stakeholders or companies.

Many field-based models exist in which agents are manipulated and vice-versa, such as in the example of the deer – biomass system. Instead of choosing between a field-based or agent-based modelling approach, often an integrated approach is taken, in which both continuous fields and discrete agents are manipulated, examples of which can be found in Bennett et al., [14], Castilla-Rho et al., [28], Sbihi et al., [150], Schelhaas et al., [151], and Schippers et al. [152].

In both field- and agent-based models, and especially in integrated field- and agent-based models, multiple kinds of representations of state variables are manipulated. Various physical data models exist for storing these kinds of state variables. APIs for storing rasters and objects are, for instance, provided by GDAL [58], HDF5 [162], netCDF4

[168], rasdaman [10], SciDB [13] and PostGIS [143]. There are important shortcomings of current options for storing state variables:

- Most physical data models support either the storage of rasters or of objects, but not both. When creating integrated models, the modeller needs to be familiar with multiple data models, store information in multiple datasets, and may have to store information like the locations in time multiple times. This results in a complex, error-prone, and inefficient workflow.
- Some physical data models do not support random access. This implies that when a certain piece of information needs to be found, possibly the whole dataset needs to be read first. This increases the runtime of models and their memory requirements.
- Existing physical data models often have limited support for storing locations in time, or only support locations in time relative to the Gregorian calendar. Forward models working with very small or long time scales, ranging from nano-seconds to millions of years, then need to resort to ad hoc conventions for storing locations in time.
- Some physical data models do not benefit from the increased amount of resources offered by high performance computing (HPC) facilities. For example, they do not have support for parallel I/O. Also, for performance reasons, some HPC facilities do not support the installation of server processes. This precludes the use of data models that require a database management system (e.g. PostGIS, rasdaman, SciDB).
- Some physical data models have limitations with respect to the number of objects that can be stored, or the size of the objects that can be stored. Ideally, the number of objects and the size of the objects should only be limited by limitations of the hardware, not by limitations of the data model.

In this chapter we tried to answer the question whether it is possible to design a unified physical data model that can be used to store different simulation model state variable representations in an integrated way. Such a data model should ideally simplify the way environmental modellers handle the storage of model state, and not suffer from the above mentioned shortcomings of current data models. Also, the data

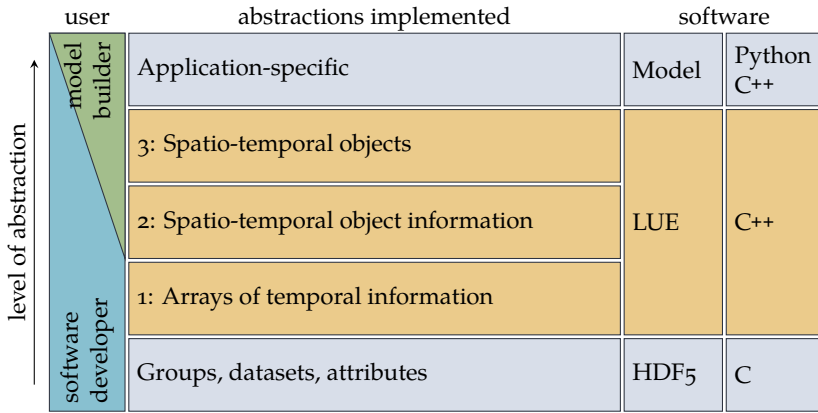


Figure 3.1: The concepts implemented in different abstraction layers of the software stack implementing the physical data model. The right columns show the names of the corresponding software implementing the abstraction layers and the programming languages used to implement them. The left column shows the expected user of the concepts in each abstraction layer. The yellow colours represent the three layers of the software stack discussed in this chapter.

model should be receptive of features currently not found in some of the popular data models, like the handling of simulated time, mobility and collections of objects.

How model state is organized in a dataset might influence the way state is represented by model variables at runtime. Whenever possible, time consuming conversions must be prevented, for example. Although in this chapter we focus on the physical data model and not on the representation of model state variables, our results align with the conceptual data model described in Bakker et al. [8], which could be used as a basis for representing model state variables.

In order to answer our research question, we first looked at the different kinds of information that are relevant to store in a unified data model. This is described in Section 3.2. The objective was to extract the commonalities from different kinds of representations of state variables and use these as a basis for our data model. This work resulted in a definition of spatio-temporal objects, which can be found in the same section. This definition of spatio-temporal objects allowed us to frame different kinds of model entities as variations of a single kind of spatio-temporal object.

Two important factors guided our data model design: performance and the management of complexity of the data model. In the case of

simulation models that read and write much data, the total runtime can become a performance bottleneck. For a data model to be relevant in environmental modelling, it must offer good performance. For this we applied the principle of locality to our data model. This is described in Section 3.3.1.

When permuting the different approaches for representing locations in time, locations in space, and properties, we initially ended up with a large number of different kinds of representations. We concluded that it was unfeasible to design a physical data model for each of these. Instead, we focused on identifying a lowest abstraction level for representing abstract temporal object information and built our data model on top of that, using abstraction levels of increasing functionality (Figure 3.1). Arrays of temporal information form the first and lowest layer of a stack of three layers of abstractions. This layer is described in Section 3.3.1. In the second layer, described in Section 3.3.2, various kinds of spatio-temporal object information are defined, in terms of the arrays of temporal information from the first layer. In the third and highest layer of abstraction, the spatio-temporal objects are defined, in terms of the abstractions in the second layer. This layer is described in Section 3.3.3. In section 3.4 we describe the implementation of our data model. Using this implementation, we were able, in a case study, to see how well different kinds of state variables from environmental models could be represented. Section 3.5 shows how this was done.

This work resulted in a physical data model for storing spatio-temporal objects, on top of the HDF5 data model (Figure 3.1). The data model is implemented in C++ and is currently exposed through a C++ API and a Python API. Appendix A contains information about how to obtain a copy of the source code.

3.2 SPATIO-TEMPORAL OBJECTS

A unified physical data model for storing state variables from environmental models must be able to represent the different kinds of representations of state variables in a uniform manner. With this we mean that it must be possible to view each kind of state variable as an example of a more general kind of state variable. For example, it must be possible to view a global variable representing growth rate for grass, a collection of multiple wandering deer (Figure 3.2a) and a single temporal biomass field (Figure 3.2b) as similar in terms of the physical data model.

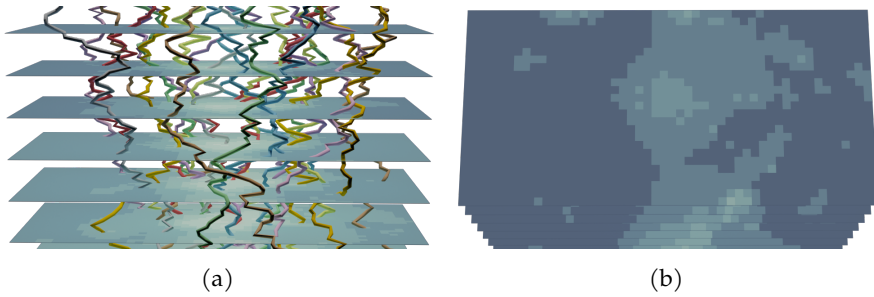


Figure 3.2: Example output of an integrated simulation model with objects (deer) and a field (environment). The third dimension, pointing up, is used here to represent time. Fields are shown for one in ten time steps. Darker coloured cells contain more biomass. (a) Locations of deer vary through time. Deer wander from cells containing less biomass to those containing more. (b) Biomass variation through time and space. Cells where deer have grazed recently contain less biomass.

In order to come up with an approach for such a unified physical data model, we first looked at the differences and commonalities between the different kinds of information represented by these state variables. In the simplest case, a state variable represents a single value that does not change during the simulation and is not located at a specific location in space. A floating point value representing the Earth's standard gravity in a plot-scale environmental model is an example of this. At the other extreme, a state variable might represent a collection of objects, each of which is located at a specific location in space and has a property value, both of which change during the simulation. A variable containing information about the locations in space and weights of a collection of wandering deer is an example of this. Besides these two extreme cases there are many more kinds of state variables, differing with respect to the size and variability of the collection of objects they represent, whether or not information changes through time or space, and the kind of property values they contain. We will now describe each of these aspects in turn, in order to end up with an approach to viewing each different kind of state variable as an example of a more general one.

We define an object as a uniquely identifiable entity whose state is simulated by a model. Objects of the same kind (for example, deer) are grouped in collections. Objects can be added and removed from such a collection (for example, deer are born and die). So, during a simulation, the set of objects that participate in the calculations may change. We

call this changing set of participating objects the set of active objects, or the active set. The active set represents *who* are active. Although we use concrete examples for objects in this discussion, we want to stress that we assume as little as possible about the nature of the objects; what a collection of objects represents is up to the application (that is: the simulation model).

Current kinds of state variables do not always represent collections of objects, as in the case with the floating point variable representing the Earth's standard gravity mentioned above. As a first generalization step towards unifying all kinds of state variables, we decided that all information represented by them must be associated with one or more objects. In the case of the Earth's standard gravity, this object represents the planet Earth.

All our objects have a presence in time and in space. With presence in time we mean that for each object, information must be stored about *when* the object was active. Objects might be active at very specific locations in time, or during longer periods. For example, it is common for objects representing living things, like deer, to be active from simulated birth to death. With presence in space we mean that for each location in time that an object was active, information must be stored about *where* the object was active. Objects might be active at specific locations in space, or in regions with a spatial extent. A bird's location can be represented by a point, and a tree's crown by a polygon, for example. Additionally, objects can be stationary, in which case the location in space is fixed through time, or mobile.

We use the concept of properties to represent *what* is present at locations in time and space that an object is active. For example, a relevant property manipulated by the model might be the weight of each deer. All objects in a collection have the same set of properties. So, either weight is stored for each deer, or weight is not stored at all.

For a single object, there are multiple ways to represent a property value, depending on the kind of property. A deer's weight can be represented by a single number, a natural park's biomass field by a 2D array, and a bird's direction and speed by a 1D Euclidean vector. The final step we took towards unifying state variables, is to represent all individual property values by nD arrays, where the dimensionality of these arrays is the same per property.

Given these generalization steps and in the context of our physical data model, we can now define spatio-temporal objects as uniquely identifiable objects for which locations in time, locations in space and

properties are stored for those locations in time that the objects where active. Our goal is that all model state variable representations, ranging from scalar constants to mobile agents with discretized properties, can be framed as collections of such spatio-temporal objects. A physical data model capable of storing such collections of objects is then capable of storing such a diverse set of state variable representations. In the next section we describe our approach to representing these objects in a physical data model.

3.3 STORING SPATIO-TEMPORAL OBJECTS

3.3.1 *Arrays of temporal information (abstraction level 1)*

Arrays of temporal information are at the lowest level of abstraction of our data model (layer 1 in Figure 3.1). They represent any kind of information for which variation through time has to be stored in the data model. At this abstraction level it is not relevant what kind of information is stored exactly. This could be the ID of an object, a coordinate in time or space, or an object's property value.

The principle that guided our approach was the principle of locality [140]. In the context of modelling, this principle states that, at any moment in time, a model accesses a relatively small portion of the data. There are two types of locality. Temporal locality means that if a model uses data, it will probably use the same data again soon. Spatial locality means that if a model uses data, nearby data (in terms of memory addresses) will probably be used soon.

The principle of locality and the fact that different kinds of memory differ a lot in speed and price, with the fastest memory being the most expensive, led to the memory hierarchy found in computers. The fastest memory is located near the CPU cores and the slowest memory is located further away, on SSD drives and spinning disks, for example. In between multiple levels of caches are available, like the caches in the storage devices themselves, the main memory and the CPU caches.

Memory is copied from lower (distant to the CPU cores) levels to upper levels in blocks of multiple values. Besides the values requested, nearby values are also copied, under the assumption that the CPU will probably need those values soon, too (spatial locality). For software to benefit from the memory caches in the memory hierarchy, it must store the values compactly, in the order in which they are accessed. That way, the caches are filled with relevant values more often. This

approach to organize the values based on what is beneficial in the context of the principle of locality and the memory hierarchy is called data-oriented design [47, 154]. The goal of data oriented design is to decrease the idle time of CPU cores, by increasing the likelihood that required values are in the nearby memory caches.

To understand what this means in the context of forward modelling, we looked at the data access patterns in forward models defined using three modelling environments: PCRaster field-based modelling environment [101], NetLogo agent-based modelling environment [173] and Mason agent-based modelling environment [115]. In these models we can typically (but not always) identify a number of nested iterations (from outer to inner):

1. *Time*: A model iterates through time until the final state of the modelled system has been calculated. Each system's state is calculated based on the system's previous state(s).
2. *Operations*: Each individual state is calculated by performing a number of operations in sequence. This set of operations is the core of the model and implements the modelled processes.
3. *Objects*: Each individual operation iterates over a set of objects to calculate some result. In this calculation, the operation typically uses a small number of object properties. An example of this is an operation that calculates the health of each deer in a collection, based on each deer's age and weight.

This iteration scheme assumes that a simulation model consists of a sequence of operations performed iteratively through time on (selections of) objects. This is a relatively simple approach, but it has some benefits we think are good to have, the most important one being that the approach naturally aligns with data-oriented design. The iteration scheme suggests a preferred ordering for the storage of information. Values for (only) the same kind of object information, like the values of a single property for all objects, should be stored close to each other (in terms of memory addresses). These collections of values should then be ordered by time. This will increase the likelihood that modelling software benefits from the caching in the memory hierarchy. When an operation iterates over a collection of objects, reading specific property values, the memory caches will likely be filled by relevant property values, not including information that is not currently needed.

Based on the principle of locality, data orientation and the iteration scheme often used in forward modelling, we decided to store information for each specific kind of object information as close together as possible and sorted by time. This means that, for example, for all objects all locations in space are stored close together, and property values of a single property are stored close together as well. This contrasts with an approach where, for each object individually, all information is stored together.

The most compact way to store multiple values in computer memory is as an array, whose element values are stored in contiguous memory locations. Therefore, we concentrated on finding ways to organize spatio-temporal object information as arrays. Each array has a shape, which is the collection of the size of each of the array's dimensions. The number of these dimensions is the array's rank. For example, a spatial raster can be represented by an array with rank two whose shape is equal to the number of rows and columns. A single array can contain multiple arrays with a smaller rank. For example, multiple spatial rasters associated with multiple locations in time can be represented by an array with rank three, whose shape is equal to the number of locations in time, rows and columns. This packing of arrays in larger arrays results in the most compact way to store object related information, which is beneficial in the light of the above mentioned memory caching.

Whether or not individual arrays containing information per object and per location in time can be packed in larger arrays depends on the shape of each individual array. Here we define object arrays as arrays containing a piece of information for a single object and for a single location in time. This could be an object's ID, or a property value, for example. We call the array resulting from packing one or more object arrays in larger arrays the value array.

Multiple object arrays can be tightly packed in value arrays when the shapes of the object arrays are equal. Object arrays containing object IDs are all $0D$ arrays that can be packed in a single $1D$ value array. Object arrays containing $2D$ arrays representing, for instance, the biomass property of multiple natural park areas cannot be packed in a single $3D$ value array.

The criteria for deciding whether or not object arrays can be packed in value arrays are shown in Table 3.1. Permuting these criteria resulted in six approaches for packing object arrays into value arrays. Here, it is not yet relevant what exactly is represented by the information in the

Do arrays contain information that changes through time?			
		Does the shape of arrays differ per object?	
		Does the shape of arrays change through time?	
no	no	no	<i>constant value × same shape array</i>
no	yes	no	<i>constant value × different shape array</i>
yes	no	no	<i>variable value × same constant shape array</i>
yes	no	yes	<i>variable value × same variable shape array</i>
yes	yes	no	<i>variable value × different constant shape array</i>
yes	yes	yes	<i>variable value × different variable shape array</i>

Table 3.1: Taxonomy of object arrays. Permuting object array value variability through time (*constant value* versus *variable value*), object array shape difference per object (*same shape array* versus *different shape array* and object array shape variability through time (*constant shape array* versus *variable shape array*) results in six kinds of object arrays. The terms in the last column are used in the text.

object arrays (for example object IDs, locations in time, or property values). What is relevant is that there are different kinds of object arrays and that the differences between them determine how to represent them in a physical data model. Our hypothesis is that with a limited set of six kinds of relatively low-level data models for storing arrays of temporal information, we can represent the much larger set of different kinds of higher-level spatio-temporal objects.

Six figures corresponding with the six object array kinds from Table 3.1 illustrate the object array kinds and their packing into value arrays. Object arrays of different objects can have the same shape (Figure 3.3, 3.5, 3.7) or a different shape (Figure 3.4, 3.6, 3.8). Note that the object arrays shown in the figures are small 1D arrays, but in reality object arrays can be very large and have a very different shape. Object arrays for temporal information can have a constant shape (Figure 3.5, 3.6) or a variable shape (Figure 3.7, 3.8).

Packing object arrays into value arrays is not possible for every kind of object array. Figure 3.8 shows that for each object array and each location in time, the object information needs to be stored in a separate value array. This potentially results in a large number of value arrays. The approach taken in this extreme case will not be beneficial for the memory caching. On the other hand, Figure 3.3 and 3.5 show that all object arrays (of all locations in time in the case of Figure 3.5) can be stored in a single value array (ordered by location in time). This approach will potentially be beneficial for the memory caching.

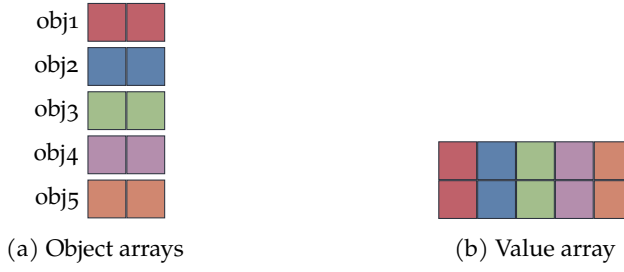


Figure 3.3: *constant value \times same shape array*: Object arrays for different objects have the same shape. Only a single object array needs to be stored per object. All object arrays are packed into a single value array. In this example, five 1D object arrays for five different objects are shown. Each object array contains two values, for example, an x and y-coordinate pair. The value array is a 2D array of shape (5,2). Applicability: object IDs, stationary space points and constant nD property values where each array has the same shape, like scalars and Euclidean vectors.

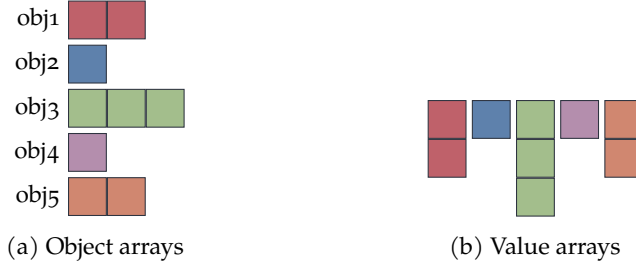


Figure 3.4: *constant value \times different shape array*: Object arrays for different objects have different shapes. Only a single object array needs to be stored per object. Each object array is packed into a separate value array. Applicability: constant nD property values where each array has a different shape, like spatial rasters for differently discretized areas.

The collection of active objects often changes during a simulation (Section 3.2). In case of the packing of the temporal object array kinds shown in Figure 3.5, 3.6, 3.7, and 3.8, this information is not part of the value arrays; it has to be explicitly stored somewhere else in the data model. For details about our approach for doing this for each of the temporal object array kinds, we refer to Appendix 3.8.1.

3.3.2 Spatio-temporal object information (abstraction level 2)

In Section 3.2 we defined spatio-temporal objects as uniquely identifiable objects for which locations in time, locations in space and proper-

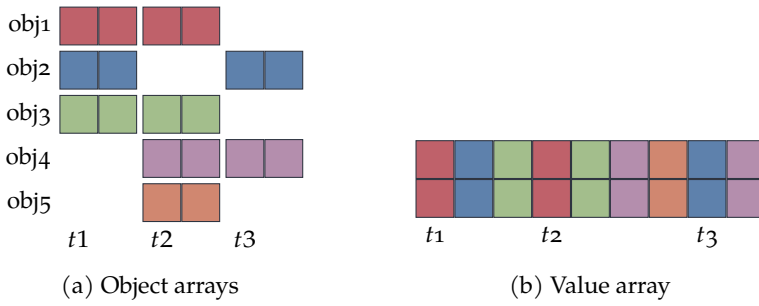


Figure 3.5: *variable value \times same constant shape array*: Object arrays for different objects have the same shape, which does not change through time. All object arrays are packed into a single value array. Object arrays with the same colour are related to the same object. Applicability: mobile space points, temporal nD property values, like scalars and Euclidean vectors.

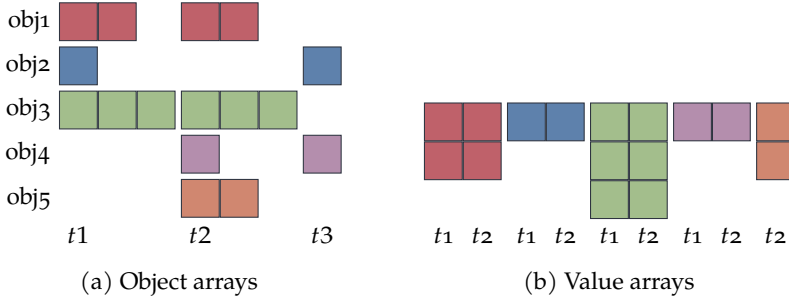


Figure 3.6: *variable value \times different constant shape array*: Object arrays for different objects have different shapes, which does not change through time. Per object, object arrays for multiple locations in time are packed into a single value array. Applicability: temporal nD property values, like spatial rasters for differently discretized areas.

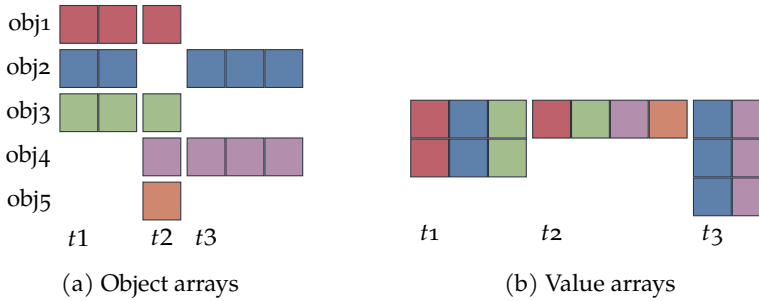


Figure 3.7: *variable value \times same variable shape array*: Object arrays for different objects have the same shape, which changes through time. Per location in time, object arrays for multiple objects are packed into a single value array. Example: temporal nD property values, like spatial rasters for equally discretized areas and for which this discretization changes through time.

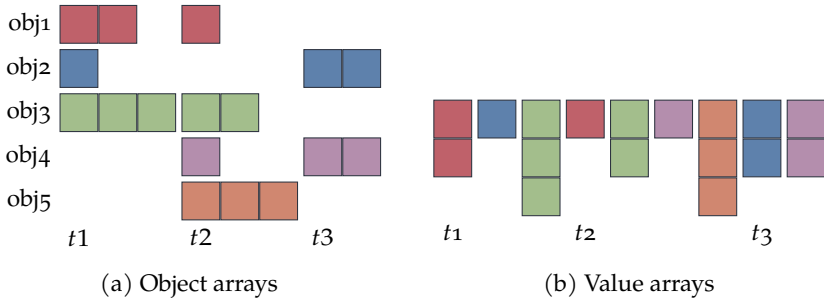


Figure 3.8: *variable value \times different variable shape array*: Object arrays for different objects have different shapes, which changes through time. Each object array is packed into a separate value array. Applicability: temporal nD property values, like spatial rasters for differently discretized areas and for which this discretization changes through time.

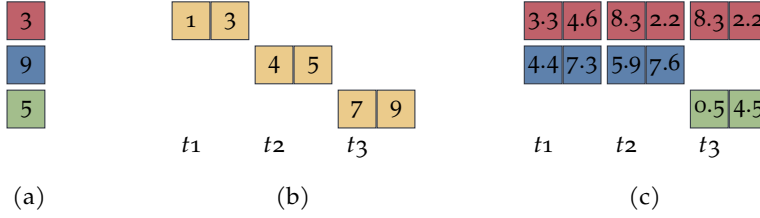


Figure 3.9: Examples of object arrays for representing spatio-temporal object information. (a) IDs associated with constant object information of three objects (see Figure 3.3 for packing). (b) Time boxes to be associated with collections of objects that are active (see Figure 3.5 for packing). (c) Mobile space points of three objects (see Figure 3.5 for packing).

ties are stored for those locations in time that the objects where active. We will now describe for each of these pieces of object-information how we decided to represent it in terms of the six kinds of object arrays described in the previous section. These abstractions correspond with the second abstraction layer implemented in our data model (Figure 3.1)

3.3.2.1 Identity

Object identity can be represented by a unique unsigned integer value which, represented as an array, corresponds with a oD array with an empty shape. This shape is the same for all objects (*same shape array*). In case of object information that does not change through time, the associated object identity can be represented by the *constant value \times same shape array* object array kind (Figure 3.9a). In case of object informa-

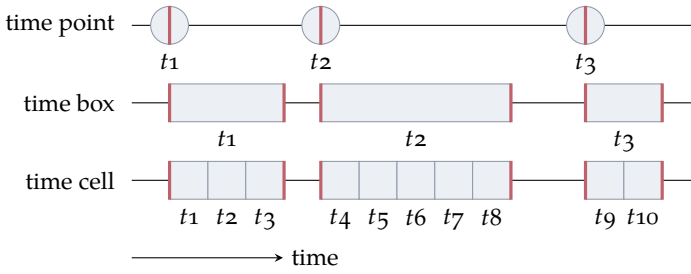


Figure 3.10: Three approaches for representing locations in time that objects can be active: three time points, three time boxes, ten time cells. The red lines are the time points that are stored in the data model. For time cells, an additional count per time box is stored.

tion that does change through time, for each location in time the IDs of the active objects can be represented by the *variable value × same constant shape array* object array kind (Figure 3.5, Appendix 3.8.1.1, 3.8.1.2 and 3.8.1.3), or by encoding the IDs of the active objects in the meta-information of the value array used to store each object array (Appendix 3.8.1.4).

3.3.2.2 Locations in time

We implemented three different approaches for storing locations in time that objects can be active: time points, time boxes and time cells (Figure 3.10). Which objects are actually active at these locations is handled by object tracking (Appendix 3.8.1). A time point is used to represent a specific location in time objects can be active. A time box is used to represent a period of time objects can be active. It is defined by a start time point and an end time point. Time cells are used to discretize time boxes for more fine grained tracking of object activity.

We represent time points by positive integral numbers representing an amount of time (duration) since an epoch. Details about this approach for handling time can be found in Appendix 3.8.2. Time durations can be represented by 1D object arrays with shape (1), and this shape does not change through time. This matches the *variable value × same constant shape array* object array kind (Figure 3.5).

Time boxes can be handled similarly as time points, but instead of storing a single time point per location in time, two increasing time points need to be stored. So, a single time box can be represented by a 1D array with two durations in it, representing the start and end time points. The shape of this array is (2) and this shape stays the

same through time (Figure 3.9b). This matches the *variable value × same constant shape array* object array kind (Figure 3.5).

For time cells, we store an additional count for each time box. Counts are unsigned integers, represented by *variable value × same constant shape array* object arrays.

3.3.2.3 Locations in space

Representing locations in space where an object is active can be done in multiple ways. For example, the Simple Feature Access standard [134], defines points, lines, triangles, and multi-polygons, amongst others. We implemented a sub-set of these approaches: space points and space boxes, which we extended to support defining locations in 1D, 2D and 3D space. Both stationary and mobile locations in space are supported.

A space point is used to represent a specific location in space where an object is active. A space box is used to represent a linear, rectangular or cuboidal region of space where an object is active. It is defined by two diagonally opposite space points.

A space point can be represented by a coordinate in each spatial dimension. A coordinate can be represented by a number (integer or floating point). The corresponding object array of a space point, then, is a 1D array with a shape equal to the rank of the space (1, 2, or 3 dimensions). For each point, this array contains the coordinates. Stationary space points are represented by the *constant value × same shape array* object array kind (Figure 3.3). Since object arrays containing space points for multiple locations in time have the same shape (the same 1D array as in the stationary case), mobile space points are represented by the *variable value × same constant shape array* object array kind (Figure 3.9c).

As with space points, a space box can be stored in a 1D object array, but this time the shape is equal to twice the shape of a single space point. The object array kinds for representing this information are the same as those for representing space points: *constant value × same shape array* object array kind (Figure 3.3) in case of stationary space boxes, and *variable value × same constant shape array* object array kind (Figure 3.5) in case of mobile space boxes.

3.3.2.4 *Properties*

In environmental modelling various kinds of property values are used to represent an object's trait. These kinds of values differ from each other based on whether or not these values vary through time, whether or not the shape of the arrays representing the values differ per object, and whether or not the shape of these arrays varies through time. For example, the weight property of each deer can be represented by a single number, which varies through time, while the surface elevation property of a natural park area can be represented by a constant or variable 2D numeric array, depending on whether the elevation changes through time.

The criteria with which property values can be classified match the ones we identified for our six kinds of object arrays, described in Section 3.3.1. In the previous sections, specific kinds of object arrays were used to represent specific kinds of object information. In the case of properties, all six object array kinds from our taxonomy can be used for storing property values. This way our data model can support a wide variety of kinds of property values used in environmental modelling.

As an example, the weight of deer through time can be stored in a *variable value × same constant shape array* object array kind (Figure 3.5). For each location in time that each deer was active, the value array will contain a floating point number representing the weight. As another example, a simulation model modelling the evolution of the elevation of the land surface of multiple research area objects can store 2D digital elevation models as *variable value × different constant shape array* object arrays (Figure 3.6, assuming the research areas are stationary and have a different, but constant shape).

Property values can be discretized through either or both time and space. We decided to store information about how property values are discretized as a property itself. In case of a spatial raster, for example, information about the number of rows and columns a 2D property value is discretized in is stored in a separate property, and linked to the property being discretized. The advantage of handling information about a discretization as a property is that this information itself can vary through time and potentially even through space. This is useful in simulation models where the spatial resolution of rasters changes through time, for example.

3.3.3 *Spatio-temporal objects (abstraction level 3)*

Since we are now able to represent the individual kinds of object information (identity, locations in time, locations in space and properties), we can focus on the representation of actual spatio-temporal objects, in which all this information is combined and can be accessed in an integrated way. The abstractions mentioned in this section are part of the third and highest abstraction level implemented in our physical data model (Figure 3.1).

One of the goals we had when designing the abstractions at this level was that it must be possible to coherently store object-related information at different (kinds of) locations in time and space. For example, for a collection of birds, it has to be possible to store information about the winter grounds, the migration route and the summer grounds. Another goal we had was that no information should be stored twice in the data model.

We identified two levels of grouping of object-related information. At the first level of grouping we combine information about which objects exist (identity), when they are active (locations in time), where each object is (locations in space) and what is present at those locations (properties). We call this level of grouping the property-set, the collection of locations in time the time domain, and the collection of locations in space the space domain. Within a property-set the information about the active sets, locations in space and properties is ordered by the locations in time. Also, within a property-set there can only be one time domain and one space domain. These domains are shared between all properties in the same set. Object-related information that is located at different locations in time or space is stored in different property-sets. For example, the constant sex of deer and their variable weight are stored in separate property-sets, having different time and space domains.

At the second and higher level of grouping, we group objects of the same kind with their property-sets. We call such a collection a phenomenon. Within a phenomenon, all objects are of the same kind, like deer or natural parks, and each object can be identified by a unique ID (identity). Also, all information related to a specific kind of objects is stored within a single phenomenon, in one or more property-sets.

This grouping of spatio-temporal object information is similar to the conceptual data model presented in Bakker et al. [8], whose design had a similar goal as our physical data model: to represent different kinds of

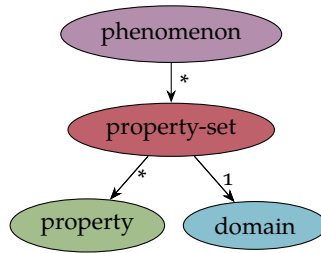


Figure 3.11: Conceptual data model for representing different kinds of state variables (adapted from Bakker et al. [8]). A phenomenon contains zero or more property-sets, each of which contains a single domain (locations in time and space) and zero or more properties.

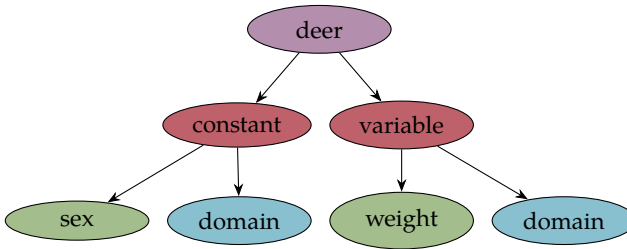


Figure 3.12: The deer phenomenon represented by the conceptual data model of Bakker et al. [8]. The colours correspond with those from Figure 3.11.

state variables in a uniform manner in order to make it more convenient for environmental modellers to create models in which these different kinds of variables are manipulated. The conceptual data model is shown in Figure 3.11. It defines *what* information is represented, and the physical data model defines *how* this information can be stored in a dataset. In an environmental modelling environment, the conceptual data model can be used as a basis for the representations, or data types, of the model state variables, whereas the physical data model can be used to allow these state variables to be persisted for later retrieval.

To illustrate the grouping of spatio-temporal object information in our data model, we will describe how to represent wandering deer. The implementation of this example is described in more detail in Section 3.5. Figure 3.12 shows the conceptual data model for a deer phenomenon. These deer are simulated using a model in which the spatial distribution of biomass in an area, and the location and weight of the deer are influenced by each other. The representation of the deer in our physical data model (Figure 3.13) is very similar to the conceptual data model. For each deer the following information needs to be

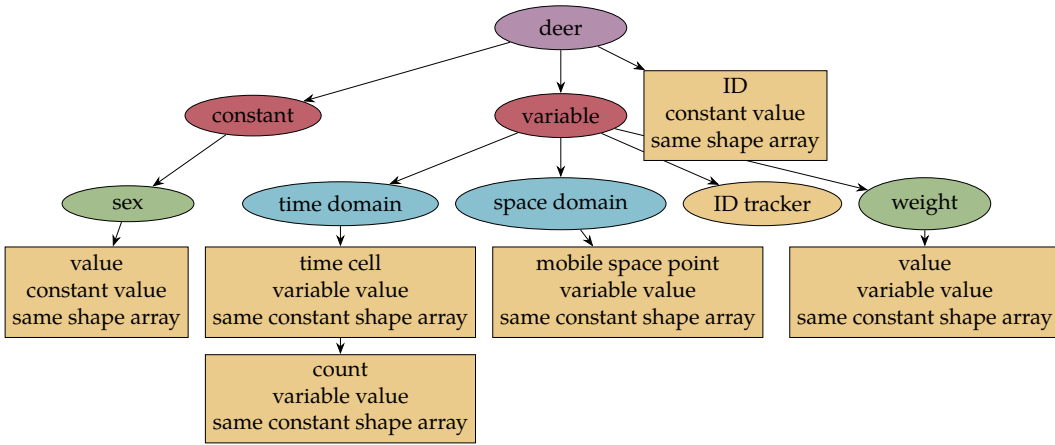


Figure 3.13: Graphical representation of the physical data model of the deer phenomenon. Most colours correspond with those from Figure 3.11. The orange boxes are collections of values represented by one of the six array kinds described in Section 3.3.1. To keep the figure readable, the object ID tracker is not expanded into arrays.

stored: a unique ID, the sex, the location in space and the weight. The ID and sex do not change through time, while the location in space and the weight do. Since within a property-set, there can only be a single combination of a time and space domain, this means that two property-sets are defined. In the first property-set, named constant, no locations in time and space are stored. For each deer, we store whether the deer is male or female in a property. In the second property-set, named variable, we use the time cell time domain kind, the mobile space point space domain kind, and a property for storing oD numeric values representing the weights. Figure 3.13 shows that all object-information is stored using one of the six array kinds described in Section 3.3.1. In this case, most information is stored using the *variable value* \times *same constant shape array* array kinds, but this depends on the specific kind of information stored. When storing rasters for differently shaped areas, for example, a property containing *variable value* \times *different constant shape array* array values must be used. And when these rasters change shape through time, a property with *variable value* \times *different variable shape array* array values must be used, instead.

3.4 IMPLEMENTATION

We implemented our physical data model using HDF5 [162]. Besides being a software library and a file format, HDF5 is a data model itself. The HDF5 logical data model can be used to organize information to be stored in a file. The most important parts of the HDF5 data model are groups, datasets and attributes. Groups are used to aggregate other groups and datasets, while datasets are used for storing multidimensional arrays. With attributes, additional information can be associated with groups and datasets. The hierarchical nature of the data model (groups can contain other groups) allows for complex nested structures to be represented. Multiple HDF5 datasets (and groups and attributes) can be stored in a single HDF5 file. HDF5 datasets can be optionally configured to be extendable along one or more dimensions.

We have defined our data model in terms of the HDF5 logical data model, using mainly groups, datasets and attributes. The implementation, named LUE¹, consists of a C++ library that implements the data model, and a Python package with which the data model is made accessible from Python scripts. In the C++ library we implemented the abstraction layers as discussed in Section 3.3 separately (Figure 3.1). We will now describe the implementation of each of these layers.

3.4.1 *Arrays of temporal information*

For each of the object array kinds described in Section 3.3.1 and shown in Figure 3.3 – 3.8, we have implemented code to store the value arrays. This low level of abstraction is implemented on top of a C++ library that wraps the HDF5 C library. For each object array kind, this layer contains code for reading and writing individual object arrays, or, when applicable, collections of object arrays. HDF5 datasets are used to store value arrays. It is possible to read or write a single 1D object array containing three floating point values, or multiple 1D object arrays in one go, each containing three floating point values, or such a collection of object arrays for multiple locations in time.

¹ LUE stands for Life, the Universe and Everything, which is the title of one of the books in Douglas Adams' Hitchhiker's Guide to the Galaxy "trilogy". Here, it refers to the fact that in designing LUE we try to make it applicable in as many contexts as possible.

3.4.2 *Spatio-temporal object information*

For all kinds of object information as described in Section 3.3.2, we developed code to read and write this information from and to a file. This code reuses the code mentioned in the previous section. Instead of reading and writing arrays of abstract temporal information, here object IDs, locations in time, locations in space, and properties are explicitly handled. The implementation contains higher level classes like `MobileSpacePoint` and `Property` which, in their implementation, use the lower level code for reading and writing arrays of temporal information. This makes it more convenient in higher level code to handle information related to spatio-temporal objects. For example, Listing 3.1 shows the declaration of a function for creating a `MobileSpacePoint` instance. Such an instance manages the I/O of space points that move through space. The function is implemented in terms of the lower level API for handling arrays of temporal information. How this is done exactly is a detail that the caller does not have to be constantly aware of.

Listing 3.1: Subset of C++ API for creating mobile space points

```
// Add mobile space point to the parent group and return
// a corresponding MobileSpacePoint instance.
MobileSpacePoint create_mobile_space_point(
    hdf5::Group& parent,
    hdf5::Datatype const& memory_datatype,
    std::size_t rank);

class MobileSpacePoint
{
    // Construct instance, based on parent group and
    // in-memory datatype of coordinates. The rank of
    // the space (1, 2, or 3) is read from file.
    MobileSpacePoint(
        hdf5::Group& parent,
        hdf5::Datatype const& memory_datatype);
};
```

3.4.3 *Spatio-temporal objects*

As described in Section 3.3.3, the highest layer of abstraction of our physical data model contains mainly grouping constructs for aggregating information about the IDs, locations in time, locations in space, and properties of the objects. We have mapped these groups onto HDF5

groups. For example, Listing 3.2 shows the declaration of a function for creating a `PropertySet` instance. Such an instance provides access to its object tracker, its time and space domains, and its properties.

Listing 3.2: Subset of C++ API for creating a property-set

```
// Add property-set to the parent group and return a
// corresponding instance. Information about the domain is
// used to setup the HDF5 constructs for writing locations
// in time and space.
PropertySet create_property_set(
    hdf5::Group& parent, std::string const& name,
    TimeConfiguration const& time_configuration,
    Clock const& clock,
    SpaceConfiguration const& space_configuration,
    hdf5::Datatype const& space_coordinate_datatype,
    std::size_t rank);

class MobileSpacePoint
{
    // Construct instance, based on parent group and name.
    // Information about the object tracker, time domain,
    // space domain and properties is read from file.
    PropertySet(
        hdf5::Group& parent,
        std::string const& name);
};
```

It is possible to combine multiple phenomena into a single HDF5 file. This makes it possible to store all simulation state in a single LUE dataset.

3.4.4 Python package

In order to make it possible to manipulate LUE data sets from Python, we implemented a Python package on top of the higher abstraction levels of the C++ library. NumPy arrays [129] are used to interface object arrays between the Python and C++ code. This makes it possible to integrate the LUE physical data model with Python applications manipulating NumPy arrays, like SciPy [86] and PCRaster [101]. The pybind11 C++ library [83] is used to expose the relevant parts of the LUE C++ API to Python and vice versa. Examples of using the LUE Python package can be found in the next section.

3.5 EXAMPLE: DEER AND BIOMASS MODEL

In this section we will describe how the state variables from a simple integrated agent-based and field-based model can be represented by our data model using the LUE Python package. We will highlight parts of the code. The complete model can be found in the source code repository associated with this manuscript (Appendix A).

The purpose of the deer and biomass model used in this example is to illustrate that our data model is capable of storing both traditional agent-based and field-based model output. In this case the deer are the agents and the natural park area with a biomass raster is the field. Example output from the model is shown in Figure 3.2. The model we used is simple and not realistic. The data is what we want to focus on here, not the validity of the model. It is possible to store more complex kinds of data than used here in our data model too, like properties with temporal discretizations, and phenomena with multiple time and space domains.

We realize that what we call a high level of abstraction can be considered a low level of abstraction by modellers. We envision that in real-world applications, an additional layer of abstraction will be built on top of our highest level of abstraction, making it easy for modellers to read and write model state information. This is shown as the upper layer of abstraction in Figure 3.1. Concepts for how this can be done have been described in Bakker et al. [8].

The following examples assume that the LUE and NumPy Python packages have been imported, a dataset is created and two phenomena are added to it (Listing 3.3). The returned `Phenomenon` instances (deer and park in this example) can be used to add property-sets to the file.

After the model has finished executing, all model state variables have been stored in a single file, formatted according to the conventions described in this manuscript. Given that each LUE dataset is also an HDF5 file, the dataset can be copied to any other platform and the information stored can be retrieved again, using the LUE APIs and HDF5 APIs.

Listing 3.3: Import required packages

```
import lue
import numpy as np

dataset = lue.create_dataset("deer.lue")
deer = dataset.add_phenomenon("deer")
park = dataset.add_phenomenon("park")
```

3.5.1 Initialize dataset

Before we can write the state of simulated spatio-temporal objects to a dataset, we must initialize the dataset. This will prepare the dataset for receiving state information.

In the deer and biomass model, the location and weight of each deer, and the distribution of biomass in the park within which the deer are located, change through time. As described in the previous text, time domains are contained in property-sets. Instead of storing multiple time domains with the same locations in time in multiple property-sets, they can be shared. This makes the resulting file smaller and prevents inconsistent time domains. In our example, the time domains of the property-sets containing temporal deer and park information can be shared. For this we create a separate phenomenon, called *simulation*, whose sole purpose is to contain a property-set with a time domain to be shared with other property-sets (Listing 3.4). We could also have shared the time domain of a property-set in the deer phenomenon with a property-set in the park phenomenon, or vice-versa.

Listing 3.4: Create property-set with time cell domain and three-hourly time steps

```
simulation = dataset.add_phenomenon("simulation")
epoch = lue.Epoch(lue.Epoch.Kind.common_era, "2019-01-01",
                 lue.Calendar.gregorian)
clock = lue.Clock(epoch, lue.Unit.hour, 3)
time_extent = simulation.add_property_set("time_extent",
                                         lue.TimeConfiguration(lue.TimeDomainItemType.cell), clock)
```

The park phenomenon in our example model contains only a single object: the park within which all the deer are located. It contains a property-set named *space_extent* with a property named *biomass* for storing the 2D biomass arrays that change through time (Listing 3.5). It also contains a property-set named *constant* with a property named *space_discretization* for storing the 1D space discretization infor-

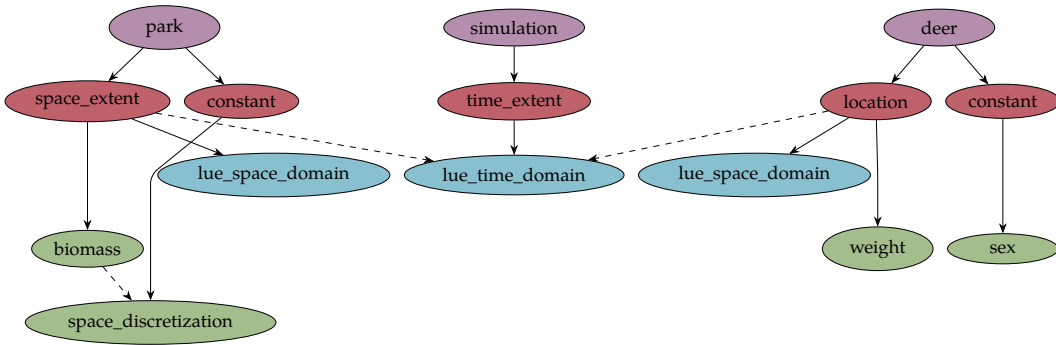


Figure 3.14: Graphical representation of the physical data model of the deer and park phenomena. The dashed lines denote symbolic links to shared information. For clarity, the collections for tracking the object IDs are not shown.

mation (number of rows and number of columns). The latter property is linked to the first to make explicit that biomass is discretized through space. The simulated biomass field is shown in Figure 3.2b.

Listing 3.5: Create park phenomenon property-sets

```

space_extent = park.add_property_set("space_extent",
    time_extent.time_domain,
    lue.SpaceConfiguration(
        lue.Mobility.stationary, lue.SpaceDomainItemType.box),
    space_coordinate_dtype=space_coordinate_dtype, rank=2)
biomass = space_extent.add_property("biomass",
    dtype=np.dtype(np.float32), rank=2,
    shape_per_object=lue.ShapePerObject.different,
    shape_variability=lue.ShapeVariability.constant)
constant = park.add_property_set("constant")
space_discretization = constant.add_property(
    "space_discretization", dtype=lue.dtype.Count, shape=(2,))
biomass.set_space_discretization(
    lue.SpaceDiscretization.regular_grid, space_discretization)
  
```

The deer phenomenon contains for each deer the sex, the location in space and the weight. Preparing the LUE dataset for receiving state information during the simulation works similar to the previous case of the park phenomenon. In the case of deer the space domain is different (mobile space points are used instead of a stationary space box) and the weight property value has a different shape (oD instead of 2D). The simulated deer are shown in Figure 3.2a. Figure 3.14 shows the resulting state of the data model.

3.5.2 Write model state

Once the dataset is initialized we can start the simulation and write the state of simulated spatio-temporal objects to the dataset. In our case, this implies writing the initial state of the biomass raster of the park, and the sex, location and weight of each deer. After that, the interaction between the biomass field and the deer agents are simulated through time, and the biomass, location and weight of each deer are iteratively updated and written to the dataset.

In Listing 3.6 the sex of each deer is written to the dataset. Given a phenomenon, property-sets can be obtained by name, and given those, properties can be obtained. Querying and assigning to property values works as if they are NumPy arrays. In this example, the value array is expanded to make space for an object array per deer. The `expand` method returns the value array on which it is called, which is then indexed for all object arrays. Assigning to these object arrays writes the new values to the dataset.

Listing 3.6: Write the sex of each deer to the dataset

```
constant = deer.property_sets["constant"]
sex = constant.properties["sex"]
sex.value.expand(nr_deer)[: ] = np.random.randint(
    low=0, high=2, size=nr_deer, dtype=np.uint8)
```

Writing state that changes through time requires also writing the object IDs of the objects for which this state is written (the active set, Appendix 3.8.1). Tracking the object IDs of the active deer is shown in Listing 3.7. Mobile space points are represented by *variable value × same constant shape array* object arrays, which implies that for tracking the IDs of the objects the index of the active set and the collection of active object IDs must be stored.

Listing 3.7: Write the object IDs of the active deer to the dataset

```
location = deer.property_sets["location"]
object_tracker = location.object_tracker
object_tracker.active_set_index.expand(1)[-1] = \
    object_tracker.active_set_index[-1] + nr_deer
object_tracker.active_object_id.expand(nr_deer)[-nr_deer:] = \
    deer_ids
```

Writing the updated location in space of active deer is shown in Listing 3.8. After the new deer locations have been simulated, additional space is created in the dataset and their coordinates are written.

Listing 3.8: Write the locations of the active deer to the dataset

```
deer_location = ...  
location.space_domain.value.expand(nr_deer)[-nr_deer:] = \  
    deer_location
```

3.6 DISCUSSION

We have designed and implemented a physical data model for simulated spatio-temporal objects. With this data model, information about the state of the simulated environment can be stored for post-processing. Although there are multiple approaches to represent state variables, depending on how they represent a value and the temporal and spatial extent and variability, we identified common aspects of these approaches, using a limited set of six kinds of object arrays (Figures 3.3 – 3.8). These object arrays and the packing of them into value arrays form the basis of our data model. Layers with increasingly more domain-specific code are defined on top of this base layer, with the layer containing the support for spatio-temporal objects at the top (Figure 3.1). Using this top-level layer of abstraction, we were able to represent some example kinds of state variables, suggesting that our data model is able to represent a diverse set of state variable kinds that otherwise would require multiple data models (Section 3.5).

Our data model provides a unified approach to storing spatio-temporal objects. Using this data model, the environmental modeller can represent multiple kinds of state variables. The modeller is not forced to use a single data model for all state variables, or use multiple very different data models. In our view, this is the main contribution of this chapter.

In our data model, information about objects is stored in multiple arrays where each array contains a single kind of object-related information. This contrasts with many existing data models, where this information is stored in database table records. As mentioned in Section 3.1, how information is organized in a physical data model might have an effect on how information is best organized in state variables in simulation models. Our data model suggests these state variables to be organized using arrays per kind of object-related information, with for the current modelled location in time the state of the current active sets of objects. This is different from the common

approach in agent-based modelling of representing state variables by class instances aggregating all object-related information.

During this work we have identified some limitations of our data model, which have to do with usability and functionality. Our data model is capable of representing complex kinds of state variables, with changing collections of mobile objects and temporal discretizations of property values, for example. Although the data model is currently usable, we think it could be made more user-friendly for simple cases by adding an additional layer of abstraction, on top of the layer implementing the storage of spatio-temporal objects. This higher level data model would represent popular existing data models, like scalars, rasters and raster stacks, and time series, and serve as an easy to use replacement for existing data models. This would allow modellers to get familiar with the LUE data model without having to learn about the lower level details first, which are necessary for the more complex cases.

With respect to functionality, some aspects are still missing from the current version of our data model, that will be added at a later stage. Examples of these are the representation of relations between objects, uncertainty in temporal and spatial locations and property values, spatial projections, discretization of presence in space, other kinds of discretizations through space, more space domain item type like lines and polygons, spatial indices, and information about topology. Whether or not the data model has to be adjusted for these additional aspects to be added remains to be seen.

Although it is of crucial importance in a high-performance computing context, we have not evaluated the performance of our data model. Since our data model is implemented in terms of the HDF5 library, the performance characteristics of this library determine the performance of our data model. To reach maximum performance we must tune our use of HDF5 for our specific use case of storing spatio-temporal objects, and probably use parallel I/O on parallel file systems. This will be the focus of future research.

3.7 CONCLUSION

In this chapter we showed the feasibility of a physical data model capable of storing different kinds of simulation model state variables in an integrated way. First, we defined spatio-temporal objects as uniquely identifiable objects for which locations in time, locations in space and

properties are stored for those locations in time that the objects were active. By assuming that absence of information about presence in time or space just means that information is valid for all simulated time or space, we were able to frame all considered kinds of state variables as spatio-temporal objects, including simple state variable kinds, like scalars or non-temporal rasters.

Next, we presented our physical data model as consisting of multiple levels of abstraction. At the lowest level, arrays of general temporal arrays are represented. On top of that, spatio-temporal object information (object identity, locations in time and space, and properties) is represented, in terms of the general temporal arrays. At the highest level of abstraction, spatio-temporal objects are represented, as aggregates of spatio-temporal object information.

The resulting data model has been implemented in the LUE software package which contains APIs in C++ and Python. We have described an example of how LUE can be used from an integrated field-based and agent-based simulation model to store various kinds of state variables in a single dataset.

3.8 APPENDIX

3.8.1 *Object tracking*

Here we describe in some more detail how we keep track of for which objects information is stored in the physical data model. This is relevant for the four object array kinds used to store temporal information. For each location in time and for each active object we store where each object array can be found. How this is done differs per object array kind.

3.8.1.1 *Variable value \times same constant shape array*

Per location in time, the value array a contains the values for all active objects (Figure 3.5). An additional array *active_object_id* is used to store the IDs of the active objects (an ID is a unique number, see Section 3.3.2.1). The order of information in a and *active_object_id* is the same. An additional array *active_set_index* is used to store the index into a and *active_object_id* where, for each location in time, the ID and object array of the first active object can be found. See also Table 3.2.

store	Append ID to <i>active_object_id</i> and object array to <i>a</i>
find	Given index of location in time, read index (<i>s</i>) of section of active objects from <i>active_set_index</i> and determine index (<i>o</i>) of object ID in <i>active_object_id</i> . The object array is located at $a[s + o]$.

Table 3.2: Procedures for storing and finding *variable value* \times *same constant shape array* object arrays.

store	Append ID to <i>active_object_id</i> and object array to a_t
find	Given index of location in time, read index of section of active objects from <i>active_set_index</i> and determine index <i>o</i> of object ID in <i>active_object_id</i> . The object array is located at $a_t[o]$.

Table 3.3: Procedures for storing and finding *variable value* \times *same variable shape array* object arrays.

3.8.1.2 *Variable value* \times *same variable shape array*

Per location in time, a separate value array a_t contains the values for all active objects (Figure 3.7). Similar to the previous case, additional arrays *active_object_id* and *active_set_index* are used to track the IDs of active objects and the offset into *active_object_id* of the ID of the first active object, respectively. Each value array a_t is named after its corresponding (index of) location in time. See also Table 3.3.

3.8.1.3 *Variable value* \times *different constant shape array*

Per object, a separate value array a_o contains the values for all locations in time that the object was active (Figure 3.6). Again, an additional array *active_object_id* is used to track the IDs of active objects. Array *active_object_index* is used to track the indices into the a_o arrays of the active objects. Array *active_set_index* is used to store the offset into *active_object_id* and *active_object_index* of the first active object per location in time. See also Table 3.4.

3.8.1.4 *Variable value* \times *different variable shape array*

Per location in time and per object, a separate value array $a_{t,o}$ contains the object array (Figure 3.8). See also Table 3.5.

store	Append ID to <i>active_object_id</i> , append index of object array in a_o to <i>active_object_index</i> , and append object array to a_o
find	Given index of location in time, read index of section of active objects from <i>active_set_index</i> and determine index of object ID in <i>active_object_id</i> . Given this index, lookup index t of object array in <i>active_object_index</i> . The object array is located at $a_o[t]$.

Table 3.4: Procedures for storing and finding *variable value* \times *different constant shape array* object arrays.

store	Create value array named after current location in time and object, and write object array to it
find	Open value array named after current location in time and object

Table 3.5: Procedures for storing and finding *variable value* \times *different variable shape array* object arrays.

3.8.2 Representing time

In order to be able to represent time points of different resolutions (like nanoseconds and centuries), we used an approach inspired by the C++ chrono library². Time points are dependent on a clock. A clock represents a period of time since an epoch, and has a certain resolution. This resolution is defined by the clock's tick period, which has a unit (seconds or years, for example) and a count (two for ticks of two seconds or years, for example). Given this, a time point can be represented by a duration, which is represented by a number of ticks.

We decided to represent a collection of time points by a single clock, represented by an epoch, a unit (a string) and a count (a positive integral number), and a duration for each time point, represented by counts (positive integral numbers). We represent epochs by an epoch-kind, and an optional origin and optional calendar (all strings). Currently supported epoch kinds are Common Era (CE) and Formation of Earth, but more can be added. The currently supported calendar is Gregorian.

² <http://en.cppreference.com/w/cpp/header/chrono>

Part II

SCALABLE COMPUTING

AN ENVIRONMENTAL MODELLING FRAMEWORK BASED ON ASYNCHRONOUS MANY-TASKS: SCALABILITY AND USABILITY

Adapted from: K. de Jong, D. Panja, M. van Kreveld, and D. Karssen-berg. “An environmental modelling framework based on asynchronous many-tasks: scalability and usability.” In: *Environmental Modelling & Software* 139 (2021), p. 104998. DOI: 10.1016/j.envsoft.2021.104998

Environmental modelling frameworks allow domain experts, rather than software developers, to implement and run numerical simulation models in earth and environmental sciences. Because of the need to use more detailed process representations or larger datasets as input to models, it may become infeasible to perform modelling studies, due to the increased amount of time it takes for models to calculate results. The objective of this study is to evaluate the asynchronous many-task approach in the implementation of a prototype scalable modelling framework. We evaluate the scalability of local, focal, and zonal map algebra operations, and an example model in which these operations are combined. Our results show that the capacity of the operations and the example model to use additional hardware, like nodes in a computer cluster, is good. With our freely available prototype framework, models can be executed faster and modelling studies processing considerably more data can be performed.

4.1 INTRODUCTION

Environmental modellers simulate the physical and biological environment using computer models. These models can be developed using a multitude of software, ranging from relatively low-level general purpose programming languages with no built-in support for environmental modelling, like C, C++, D, Fortran, Java and Rust, to high-level modelling frameworks containing pre-built model building blocks, like Google Earth Engine for earth science data and analysis [64],

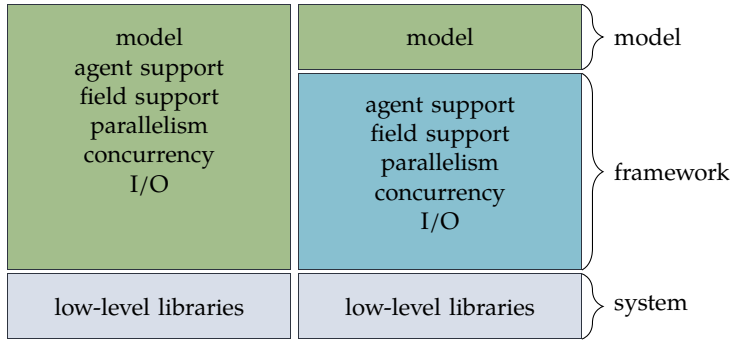


Figure 4.1: Two stacks of model and support code. A model developer writing a model from scratch has to write more code (left) than a developer using a modelling framework (right). The green boxes represent the amount of code the model developer has to write. The blue box represents the framework code.

MATLAB [80], the NetLogo agent-based modelling framework [173], and the PCRaster field-based modelling framework [101]. An advantage of using modelling frameworks is that they, in different degrees, hide some of the low-level complexities of implementing models. This speeds up model development and allows domain experts without a background in software development to develop models (Figure 4.1 and [96]).

Some of the model development interfaces are inspired by map algebra [166], which is also the approach that will be followed here. Existing examples using map algebra include the Python programming language packages provided by PCRaster, ArcGIS, and QGIS. In map algebra, fields of spatially varying environmental attributes are represented by rasters, which can be combined and translated into new rasters using a procedural programming style. A set of such translations, simulating environmental processes during a single time step, can be used by a modelling framework to do forward iteration through time, error propagation, and data assimilation [101]. The framework provides the elementary data structures and modelling algorithms used by modellers in their models. Ideally, models built with such a framework offer good performance, whatever the combination of modelling operations used. In our study we look at designing and building such a framework for developing environmental models.

Over time, models often outgrow their capacity to calculate results in a timely manner. This may be because of an increase in dataset sizes used by models, an increase in temporal or spatial resolution or extents,

or an increase in model complexity. In order to solve this discrepancy between the size and performance requirements of large models, and their capacity to provide results, models must increase their ability to use the current generation of hardware. In general, newer generations of hardware are more powerful, but also more complex, than earlier generations. Typically, current computers contain more cores, more kinds of cores, and a deeper memory hierarchy. Additionally, the availability of computer clusters to modellers, containing multiple compute nodes connected by low-latency network connections, has increased. The challenge we focus on is that of building a modelling framework that makes better use of the available hardware.

There are multiple approaches for developing a modelling framework implementing a collection of parallel and distributed map algebra operations. An intuitive and popular approach to creating parallel versions of such operations is to use the synchronous fork-join paradigm, supported by OpenMP [39] for example, in which individual algorithms implementing these operations are parallelized and called in sequence. Examples of frameworks using this approach are the Parallel Raster Processing Library (pRPL) [70], the Parallel Raster-based Geocomputation Operators (PaRGO) [146], and the Parallel Cartographic Modeling Language (PCML) [155]. A drawback of this approach is that it introduces implicit synchronization points. At least at the end of each operation, the flow of control will wait for all tasks to finish before returning to the caller, resulting in workers (like CPUs and GPUs) being inactive for some time. This negative effect of synchronization points increases with the number of workers and the degree of load imbalance between the workers. Note that load imbalance between workers is common in environmental modelling operations, resulting from an uneven spatial distribution of no-data values, or because of high spatial dependencies between cell values, as is the case in some operations that operate on a flow direction network.

An alternative approach for implementing a collection of parallel and distributed map algebra operations is to use asynchronous many-tasks (AMT). One of the advantages of this approach is that it avoids unnecessary synchronization points. With AMT, work to be done is encoded in a set of relatively small tasks with data dependencies among them. Tasks are spawned asynchronously, allowing the main flow of control to continue into multiple modelling operations, resulting in more tasks being spawned. Tasks get scheduled on workers after their inputs have become available. This approach results in a larger

collection of tasks than is possible when parallelizing algorithms individually, as in the case of the synchronous fork-join approach. The advantage of having a larger collection of runnable tasks is that it decreases the chance of workers being inactive. Examples of runtime systems that support AMT on distributed memory systems are Chapel [30], X10 [31], HPX [91] and Charm++ [94], of which the first two are specific languages and the latter two are software libraries. For a taxonomy of task-based parallel programming technologies see Thoman et al. [163]. We will use the AMT runtime system implemented by the HPX software library for implementing our environmental modelling framework.

The objective of this research is to evaluate the use of AMT for the development of a modelling framework containing implementations of map algebra operations, that can be used on all kinds of commodity hardware in use by the modelling community (ranging from laptops, desktops, to computer clusters). The main question we want to answer is whether the use of such a framework results in scalable models. For this we perform different kinds of scaling experiments over different kinds of workers. Scalability of models is determined by both the software implementing the compute part and the I/O part. In this study we focus on the compute part. Additionally, we review the resulting framework in terms of its usability by model developers, and we review the use of AMT in the implementation.

This chapter is organized as follows: in Section 4.2 we describe the approach of developing environmental models using map algebra in more detail; in Section 4.3 we provide more information about AMT and the HPX implementation thereof; in Section 4.4 we describe how we used AMT to implement a map algebra development interface on top of modelling algorithms; in Section 4.5 we present results of scaling experiments we performed with individual algorithms and an example model simulating wildfire, in which some of the implemented modelling algorithms are combined. We end this chapter with a discussion of the results in Section 4.6.

The AMT runtime system implemented in HPX enables us to write an initial set of high-level modelling algorithms that can be called from a map algebra-like model development interface in a modelling framework. Implementing algorithms in terms of asynchronous tasks that translate asynchronously produced input data into output data, results in a flexible system in which modelling algorithms can be combined in any order, according to the model, and still offer good scalability.

The framework developer is responsible for defining tasks and the dependencies between them, and is relieved of the responsibility of scheduling tasks and explicitly sending messages in between processes. The framework implementation is freely available for inspection and use (Appendix A).

4.2 MODEL DEVELOPMENT USING MAP ALGEBRA

Originally, the map algebra language was designed for creating cartographic models, where the models were collections of maps [166]. The language consisted of a specific set of relatively simple generic operations that translate raster data. A combination of such operations could be used to, for example, determine suitable locations for land development. The advantages of using map algebra are that a finite set of generic operations can be used to handle multiple use cases, and that it provides a level of abstraction that makes it suitable for users without a background in software development.

The principles behind cartographic modelling using map algebra have been extended towards forward numerical simulation of environmental processes as well [44, 101]. A map algebra-like language is then used to define the initial state of the modelled environmental system and to define the state transitions over time. In this context, the model refers to the code, not to the collection of maps. The model shown in Listing 4.1 is an example of an environmental model, implemented using a map algebra-like language, simulating wildfire. We used it in our experiments (Section 4.4). Outputs from the model are shown in (Figure 4.2).

Listing 4.1: Pseudocode of model simulating wildfire.

```
# ----- Initialize state -----
# Burning locations: area that is burning
fire = uniform(clone, 0.0, 1.0) < 1e-5

# Currently burning
burning = copy(fire)

# Two areas that differ in how fast they
# catch fire
kernel_51x51 = circle_kernel(25, 1.0)
burnability =
    focal_mean( uniform(clone, 0.0, 1.0), kernel_51x51) < 0.5

# Probabilities for catching fire
```

```

ignite_probability = where(burnability, 0.05, 0.01)

# Probabilities for jump fire
spot_ignite_probability = ignite_probability / 50.0

fire_age = array_like(fire, 0)
nr_burnt_cells = where(fire, zonal_sum(1, fire), 0)

# ----- Transition state -----
for t in range(nr_time_steps):
    # Find cells where at least one
    # neighbour is burning and that
    # themselves are not yet burning
    # or burnt down
    kernel_3x3 = circle_kernel(1, 1.0)
    cells_not_burning_surrounded_by_fire =
        focal_sum(burning, kernel_3x3) > 0 and not fire

    # Select cells that catch new fire
    # from direct neighbours
    new_fire = cells_not_burning_surrounded_by_fire and
        uniform(clone, 0.0, 1.0) < ignite_probability

    # Find cells that have not burned
    # down or at fire and that have fire
    # cells over a distance (jump
    # dispersal)
    kernel_5x5 = circle_kernel(2, 1.0)
    jump_cells = focal_sum(burning, kernel_5x5) > 0 and not fire

    # Select cells that catch new fire by
    # jumping fire
    new_fire_jump = jump_cells and
        uniform(clone, 0.0, 1.0) < spot_ignite_probability

    # Currently burning or previously
    # burned
    fire = fire or new_fire or new_fire_jump

    # Age of fire in timesteps
    fire_age = where(fire, fire_age + 1, fire_age)

    # Number of cells that are burning or
    # have burnt
    nr_burnt_cells = where(fire, zonal_sum(1, fire), 0)

    # Burning cells
    burning = fire and fire_age < 30

    # Classify cells in burning, burnt or

```

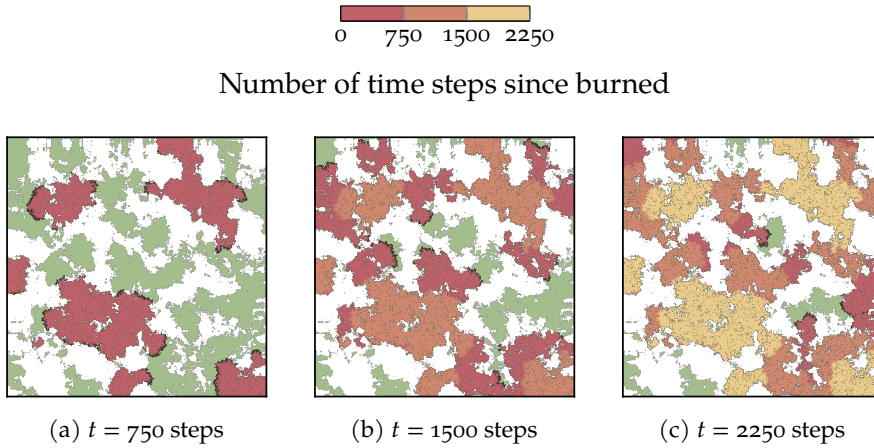


Figure 4.2: Output maps of the example model simulating wildfire for three time steps. Black cells represent cells that are burning (fire front). Green cells represent the area with the highest susceptibility for being burned. The other colours represent the age of burned cells in number of time steps, with red cells being burned most recently. The area shown is 500×500 cells.

```
# not burnt yet
state = where(burning, 1, where(fire, 2, 3))
```

Map algebra operations are often classified according to the kind of neighbourhood from which input raster cells are selected that contribute to the calculation of output raster cells. In this study we consider three kinds of operations [24, 101, 166]: local operations (Figure 4.3a), focal operations (Figure 4.3b), and zonal operations (Figure 4.3c). Operations not considered in this chapter include global operations, which can be seen as a subset of zonal operations, and network operations, operating on a flow direction network.

4.3 ASYNCHRONOUS MANY-TASKS AND HPX

The AMT programming model supports defining relatively small tasks of work that need to be executed, and the dependencies between them. The tasks and their dependencies form a directed acyclic graph that is used by the AMT runtime system to determine the order in which the tasks must be executed, and to determine which tasks can be scheduled to execute concurrently. Given enough hardware resources, the runtime system will execute concurrent tasks in parallel.

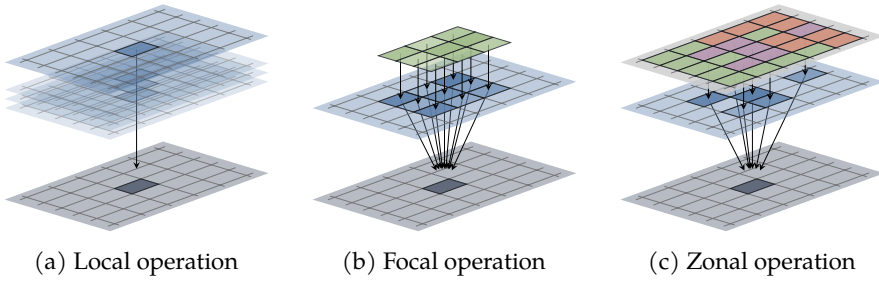


Figure 4.3: a: Each output cell is a function of input cells at the corresponding location in one or more input rasters. Example: $ph = -\log(hydronium)$. b: Each output cell is a function of input cells within a neighbourhood of input cells at and around the corresponding location in an input raster. In principle, these neighborhoods can have any size and shape, but they are often square or round and small. Example: $smooth_ph = focal_mean(ph, kernel)$. c: Each output cell is a function of input cells sharing the same class according to a second input raster. Here, the blue cells at the same locations as the purple cells contribute to the output cell shown. Example: $max_ph = zonal_max(ph, soil)$.

The requirement for the runtime system to always be able to schedule tasks for execution, is that there are enough tasks defined and few dependencies between them. In order to achieve this, tasks are created asynchronously, and do not depend on more tasks than necessary. An asynchronously created task is spawned off from its operating system (OS) thread, which continues doing other work, for example spawning off more tasks.

HPX is an implementation of the AMT programming model and runtime. It is an open source software library written in portable C++11/14/17/20 code and does not depend on a compiler from a specific vendor or on compiler extensions. It has been used to implement parallel software successfully in multiple studies [73–75, 103].

Using HPX the developer of an environmental modelling framework can define tasks and their dependencies in the usual imperative style of programming in C++. Using the HPX API, the graph of tasks is built implicitly and does not need to be explicitly managed by the developer. The framework developer’s main responsibility is to correctly represent the total amount of work to be executed by a collection of tasks and their data dependencies. The size of each task is measured in terms of its latency, which depends on the amount of data processed by the task, the number of computations performed, and on latencies involved in accessing the data. The ideal task size is large enough for the overheads of parallelization to be amortized over the sum of the

latencies of all tasks, and small enough to provide the schedulers with enough concurrent tasks to schedule on workers [69]. Since the latency of tasks is partly dependent on aspects that are only known at runtime, like data values and hardware characteristics, it is important that the task size can be influenced by the user. One way to do this is to support a parameter representing the amount of data processed by individual tasks.

To illustrate the differences between the AMT approach and other approaches to writing a modelling framework, we assume a model exists, similar to the map algebra model simulating wildfire shown in Listing 4.1, that calls three modelling operations from the framework. For simplicity, we will ignore the overheads of parallelization. A serial framework executes these operations one after the other on a single worker (Figure 4.4a). The latency of this program is the sum of the latencies of all the work that needs to be done. Since only one worker is used by this program, adding more workers will not decrease its latency. When the three operations are independent from each other, they can be executed in parallel (Figure 4.4b). The program's latency is determined by the operation taking the most time to finish. Since each operation is executed by one worker, adding more workers will not decrease this program's latency. In the implementation OS threads can be used, for example, to spawn threads doing work on multiple CPU cores. When the three operations contain concurrent tasks that can be executed in parallel (Figure 4.5), another approach can be taken. In this case, the operations are still executed one after the other, but they are partly executed in parallel (Figure 4.4c). This program's latency is determined by the sum of the latencies of the serial regions and the parallel regions. Adding more workers will not decrease the latency of the serial regions but, given enough concurrent tasks, may decrease the latency of the parallel regions. In this example it will not, though, since none of the parallel regions has more than three concurrent tasks. In the implementation, OpenMP [39] can be used for example, to create parallel regions in which multiple OS threads are used to execute tasks on multiple CPU cores.

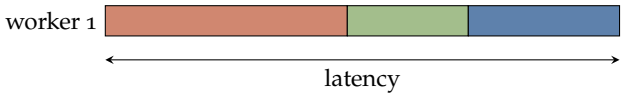
When using the AMT approach, concurrent tasks from all three operations are executed in parallel, taking the dependencies between the tasks into account (Figure 4.4d). The latency of the program is determined by the maximum of the sums of the latencies of the tasks per worker. Given enough concurrent tasks, adding more workers will decrease the program's latency. Because tasks from multiple operations

are considered there are more options to avoid load imbalance between workers.

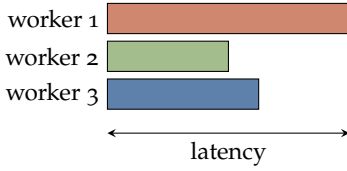
In environmental models, most rasters processed by the modelling operations depend on each other: the output rasters from operations are used as input in other operations. It is therefore unlikely to find many modelling operations whose tasks are completely independent from each other, as is shown in the idealized example in Figure 4.4d. But, since tasks are created asynchronously in AMT, tasks from different modelling operations can be scheduled for execution, as long as the input data of each of these individual tasks is ready. Depending on the modelling operation, input data of individual tasks can be relatively small subsets of the full input rasters of the operations. For example, in a model containing multiple local operations, tasks from every operation may be executing at the same time, even though output rasters from some of these operations is input of others. The AMT runtime considers individual tasks, not operations.

In HPX a data structure called *future* exists which represents the output of a task. This output may be ready to be used, or it may become ready later on. Dependencies between tasks are defined by attaching tasks to futures output from other tasks. Once a task is finished, its output future is marked ready and dependent tasks are notified. The HPX runtime manages task schedulers (one per OS thread) that manage multiple queues of tasks, some of which are ready to be executed, while others are still waiting for input dependencies to be satisfied.

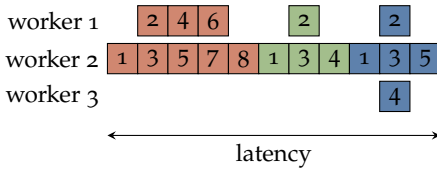
When spawning HPX tasks, the framework developer has to specify the target each task must execute on. Common targets are OS processes and object instances within processes, called components in HPX. Processes and components can be local to the computer on which a task is spawned, or remote. This is transparent to the software developer. When using the HPX API, the developer programs a single abstract machine consisting of one or more processes running on one or more computers. Because of this, HPX can be used transparently for parallel computing on both a single shared memory computer and on multiple distributed memory computers. This is an advantage over existing popular approaches that use multiple APIs, like using MPI [123] for the distribution and OpenMP for the parallelization of work.



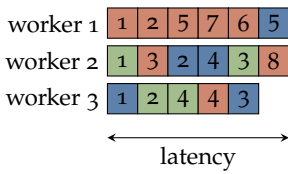
(a) Serial execution.



(b) Parallel execution using few tasks.



(c) Parallel execution using fork-join.



(d) Parallel execution using many tasks.

Figure 4.4: a: Serial execution of three operations by one worker. Each colour represents an operation. Latency is the duration of executing all the work. b: Parallel execution of three operations by three workers. c: Serial execution of three operations, but parallel execution by three workers of the concurrent tasks within each operation (Figure 4.5). d: Parallel execution by three workers of concurrent tasks of all operations.

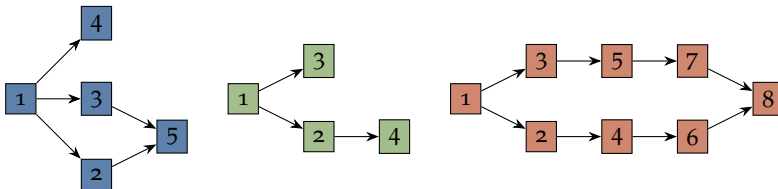


Figure 4.5: Each operation has concurrent tasks that can be executed in parallel. Each colour represents an operation and each numbered box represents a task. The length of each task represents its latency which, for simplicity, is assumed to be constant.

4.4 METHOD

4.4.1 *Implementation*

An implementation of map algebra requires a data structure for representing rasters, and operations translating input rasters to output rasters. We designed a partitioned multidimensional array data structure with two capabilities that are important for our purposes. First, the size of the partitions is configurable, which is important because it influences the size of tasks translating array partitions. Second, the partitions can be distributed over multiple operating system processes, which is important because tasks translating array partitions are sent to the data. The distribution of partitions therefore determines the distribution of most of the computational load.

Array partitions are implemented in terms of HPX component clients. These are light-weight objects providing a convenient API for interacting with, possibly remote, component server instances, containing the actual array partition elements. HPX component client objects are semantically equivalent to futures. They refer to data that may or may not be ready to use yet, but as any HPX future, they allow a task to be attached to them, which will be scheduled for execution once the data has arrived and the future becomes ready. In Figure 4.6 an example of a partitioned array is shown, whose partitions are distributed over three nodes in a cluster. The partitioned array allows the whole raster to be stored in the memory available to a single process or distributed over multiple processes, possibly on multiple nodes. The client code using partitioned array instances, like modelling algorithms, does not need to make a distinction between these two cases.

Given the partitioned array data structure, we developed algorithms implementing a set of local, focal, and zonal operations. The goal of the algorithms is to generate and distribute tasks that will perform the necessary calculations in such a way that the computational load is evenly balanced over the available processes. They are fully asynchronous. None of the algorithmic steps block the flow of control. Each algorithm may finish executing even before any of the input array partitions are available.

Input partitions may be the result of another asynchronous task, like an I/O operation reading partition data from a data set, or another local, focal, or zonal operation. This means that the array elements in such a partition may not be available yet. The idea is to attach the

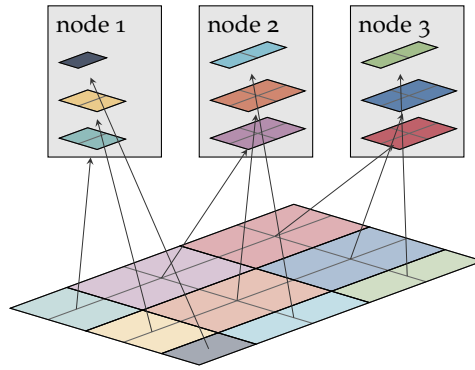


Figure 4.6: A partitioned array instance is stored in a single process and contains an array of partition client instances. Each of these refers to a component server instance actually containing the array elements, and which is possibly located in a different process. Each partition’s elements are ready to be used, are currently being calculated, or will be calculated in the future.

task for translating input elements to output elements to the future representing the input array partition. HPX will schedule such a continuation automatically for execution, once the input partition data becomes available.

Our algorithms always contain parts that execute in the process where the input partitioned arrays are located, and parts that execute in the processes where the input array partitions, referenced by the input partitioned arrays, are located. These latter parts are used by the former part to perform most of the computations. The main steps of the algorithms implementing local, focal, and zonal operations are shown in Algorithm 1, Algorithm 2, and Algorithm 3, in Appendix 4.7.1. Although some of the steps may suggest that the flow of control blocks, this is not the case. For example, when an input partition of a task is not ready yet, or partition data has not arrived yet, the flow of control will continue, generating more tasks. As soon as partitions do become ready, or data arrives, the state of associated tasks is changed (by the HPX runtime) from staged (waiting for dependencies to be satisfied) to pending (ready to run). From then on these tasks can be scheduled for execution.

The latencies involved in requesting data from an array partition depend on the location of the partition server relative to the partition client. If they are located in separate processes, latencies are (much) higher, because memory has to be copied from the server’s process to the client’s one, possibly involving network traffic. Instead, if they are

located in the same process, no memory is copied, only the address of the data is. Because input array partitions are never changed themselves, the partition data can be assumed to always be in a valid state, and no synchronization primitives, like mutexes and locks, are needed to enforce that.

The result of our approach is that calling multiple local, focal, and zonal operations after each other creates many tasks for the HPX runtime to schedule for execution, once their input data requirements are met. As long as there are more tasks that are ready to be executed than there are workers to execute them, the hardware will be fully occupied. The creation of a model's tasks will generally finish before the tasks themselves, at which point the execution of a model will block until the last task has finished executing.

4.4.2 *Scalability and performance*

To characterize the scalability of our modelling framework, we developed an example model, simulating wildfire (Listing 4.1). This model is based on concepts from existing fire models (e.g. [32, 53, 112, 167]) and its scalability and performance can thus be representative for this type of environmental models. The model is implemented by combining local, focal, and zonal modelling operations from our framework. Two processes relevant in fire models are represented by the model. The first is surface fire, where an area catches fire because it contains burnable material and a neighbouring area is already burning. The second process represented is spotting fire, where an area catches fire because an area further away is burning. The example model serves as a typical use-case for an environmental modelling framework containing map algebra operations.

Results of the scaling experiments of the example model provide information about the usefulness of AMT in the implementation of a modelling framework. We also assessed the scalability of individual local, focal, and zonal operations. Results from these experiments are useful to detect scalability issues with a specific (kind of) modelling operation.

We performed scaling experiments on a partition of a computer cluster. The hardware and software platform of each of the nodes in this partition is listed in Table 4.1. In each of the cluster nodes, CPU cores are grouped into NUMA (non-uniform memory access) nodes. Main memory is distributed over these NUMA nodes, and CPUs can

CPU	2 AMD EPYC 7451 (2 packages)
NUMA nodes	8 (4 / package)
Cores	48 (6 / NUMA node)
Clock frequency	2.3 GHz
L1d / L1i	32 / 64 KiB / core
L2	513 KiB / core
L3	8 192 KiB / 3 cores
RAM	256 GiB (32 GiB / NUMA node)
OS	CentOS 7
GNU GCC	version 10.2.0
HPX	version 1.5.0 [92]

Table 4.1: Hardware and software platform of nodes used in experiments. All nodes are interconnected with InfiniBand.

reference values stored in memory of their own NUMA node faster than values stored in the memory of neighbouring NUMA nodes. This is relevant when designing scaling experiments. When scaling over CPU cores it matters in which NUMA node these cores are located. Randomly picking CPU cores results in non-reproducible scalability measures.

We performed separate scaling experiments over three kinds of workers: 1) over the 6 CPU cores within a single NUMA node, 2) over the 8 NUMA nodes within a single cluster node, and 3) over 12 cluster nodes within a cluster partition. For smaller problems scalability over CPU cores is relevant, and for increasingly larger problems the scalability over NUMA nodes and cluster nodes is. When scaling over the 6 CPU cores, a single process was assigned to a single NUMA node and CPU cores were assigned in order from within this NUMA node. When scaling over the 8 NUMA nodes within a cluster node, as many processes were used as NUMA nodes used by each specific run, each of them assigned to the CPU cores within a separate NUMA node. When scaling over cluster nodes, on each node 8 processes were used: one per NUMA node. Using a process per NUMA node is a convenient way to make sure memory allocations and references are resolved by the nearby main memory, in the same NUMA node as the process.

We calculated both the relative strong and weak scaling efficiencies. The relative strong scaling efficiency provides information about how well the modelling framework is able to use additional workers, while the total problem size (the number of cells in the rasters processed by

worker	individual operations		model	
	strong	weak	strong	weak
core	10 000 × 10 000	4 000 × 4 000	5 000 × 5 000	2 000 × 2 000
NUMA node	30 000 × 30 000	10 000 × 10 000	15 000 × 15 000	5 000 × 5 000
cluster node	96 000 × 96 000	30 000 × 30 000	48 000 × 48 000	15 000 × 15 000

Table 4.2: Sizes of raster maps used in the scaling experiments. In case of strong scaling, the same array sizes are used for all numbers of workers. In case of weak scaling, the array sizes given in this table are multiplied by the number of workers used in the experiment.

the model) is kept constant. It is calculated by dividing the software's latency $T_{S,1}$ on a single worker by the latency $T_{S,P}$ on P workers, multiplied by P (Equation 4.1). In the case of linear scaling $P \times T_{S,P}$ equals $T_{S,1}$. In that case, doubling the number of workers halves the latency.

$$\text{strong scaling efficiency} = \frac{T_{S,1}}{P \times T_{S,P}} \times 100\% \quad (4.1)$$

The relative weak scaling efficiency provides information about how well the modelling framework is able to use additional workers, while the problem size per worker is kept constant. It is calculated by dividing the software's latency $T_{W,1}$ on a single worker by the latency $T_{W,P}$ on P workers (Equation 4.2). In the case of linear scaling $T_{W,P}$ equals $T_{W,1}$. In that case, doubling the number of workers (and the number of cells in the rasters processed) does not influence the latency.

$$\text{weak scaling efficiency} = \frac{T_{W,1}}{T_{W,P}} \times 100\% \quad (4.2)$$

Before performing the scaling experiments of the operations and the case-study model, we first determined their optimal task size, determined by the array partition size. We measured the optimal task size given the maximum array size and number of workers as used in the strong scaling experiments. In order to determine the variability in the latencies of model runs, these experiments were repeated three times. In this study we focused on the scalability of the computations. Time spent on I/O was not taken into account when measuring latencies.

In Table 4.2 the size of the arrays used in the scaling experiments are shown. The sizes were chosen such that in all experiments all CPU cores would have a relatively large amount of work to perform which

would still fit in the memory of the NUMA node. For comparison, a raster of $96\,000 \times 96\,000$ cells can cover an area as large as Australia with 30×30 m cells. Operation experiments were simulated for 500 time steps, calling the operation once for each time step, and the example model experiments were simulated for 250 time steps. These counts were chosen such that each model would take between half an hour and three hours to finish.

In Appendix 4.7.2 the pseudo code can be found of the “models” used in the scaling experiments for individual kinds of map algebra operations. The wildfire model we used is shown in Listing 4.1.

Because scalability is the main focus in this chapter, we have not optimized our code for performance. We did measure throughputs, to get an impression of the performance of each experiment. Here, throughputs are a measure of how many raster cells are being calculated per second during each experiment, assuming each experiment results in a single raster at the end of each time step. Additionally, to get an impression of the absolute performance of our framework when using a single CPU core, we compared the latency of the example model with the same model implemented using the PCRaster modelling framework [101].

4.4.3 Usability

Besides the scalability aspects of the new modelling framework we evaluate how easy the framework can be developed, and how well the resulting software can be used by model developers. To characterize this we evaluated the resulting source code with respect to one of the aspects that are generally considered an important characteristic of maintainable code, namely whether or not the code is modular and contains clearly separated layers of abstraction [82]. This is not meant to be a complete software quality analysis, but an evaluation of some of the implications of using the AMT programming model as implemented in the HPX library in the implementation of a modelling framework. To characterize the usability of the framework by model developers, we review what the implications are for the modeller to develop a model using our framework. Ideally, there should be no difference between using our framework and comparable alternatives.

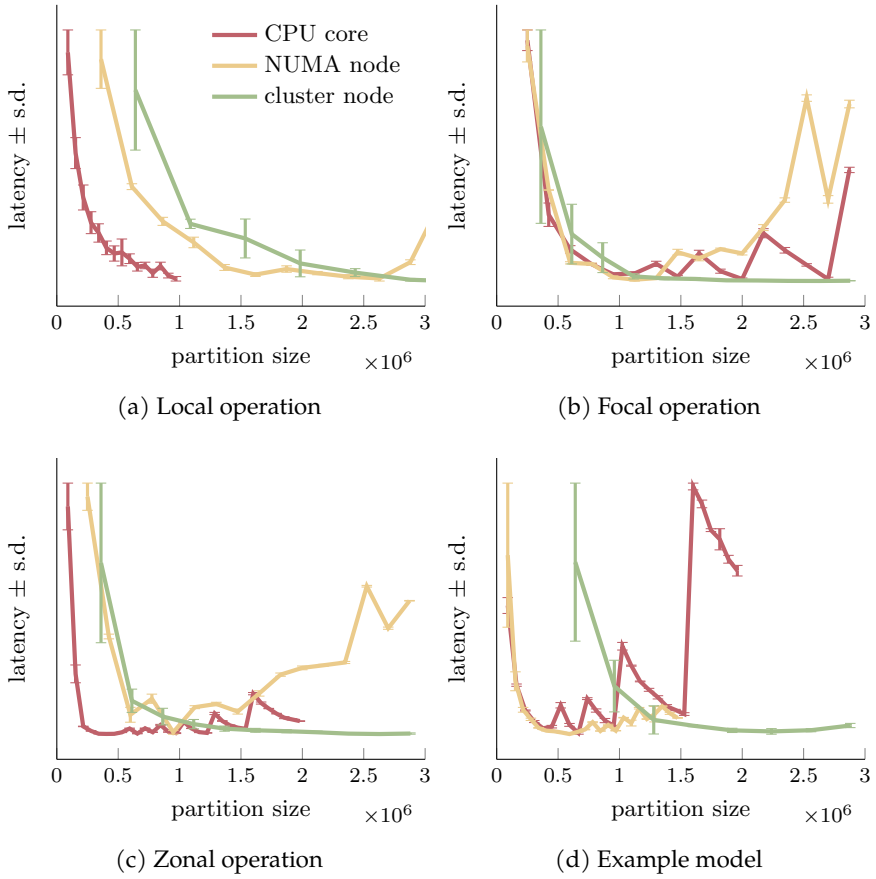


Figure 4.7: Latency of experiments for different kinds of workers and partition sizes. Partition sizes shown are the number of cells in square partitions.

4.5 RESULTS

4.5.1 Scalability and performance

The results of the partition shape experiments show that there is often a range of partition sizes that result in relatively small latencies (Figure 4.7). To provide the HPX schedulers within each process with as many tasks as possible, we selected the smallest optimal partition size to use for the strong and weak scaling experiments (Table 4.3). The experiments also show that the variability in latencies is relatively small at optimal partition sizes (Figure 4.7). We therefore did not perform the strong and weak scaling experiments multiple times.

worker	local	focal	zonal	model
core	1 000 × 1 000	1 000 × 1 000	600 × 600	650 × 650
NUMA node	1 300 × 1 300	1 000 × 1 000	975 × 975	750 × 750
cluster node	1 900 × 1 900	1 200 × 1 200	1 600 × 1 600	1 500 × 1 500

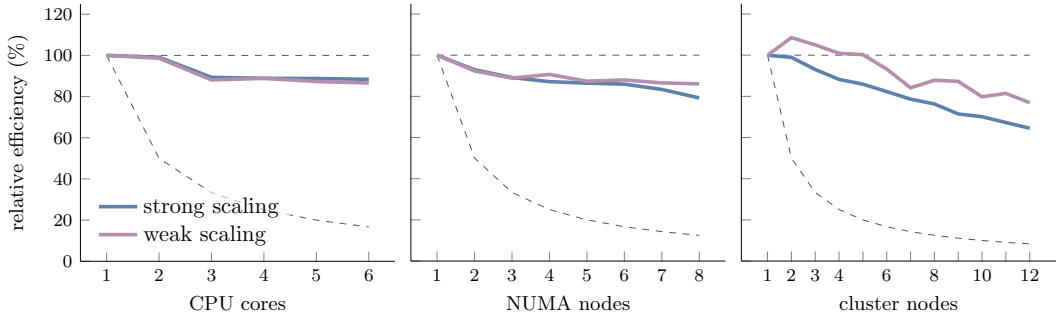
Table 4.3: Sizes of partitions used in the scaling experiments. These sizes correspond with the smallest sizes in Figure 4.7, where the latencies are small.

workers	strong scaling				weak scaling			
	local	focal	zonal	model	local	focal	zonal	model
6 CPU cores	88%	81%	87%	79%	87%	80%	86%	80%
8 NUMA nodes	79%	91%	113%	77%	86%	83%	92%	74%
12 cluster nodes	65%	54%	70%	42%	77%	57%	80%	70%

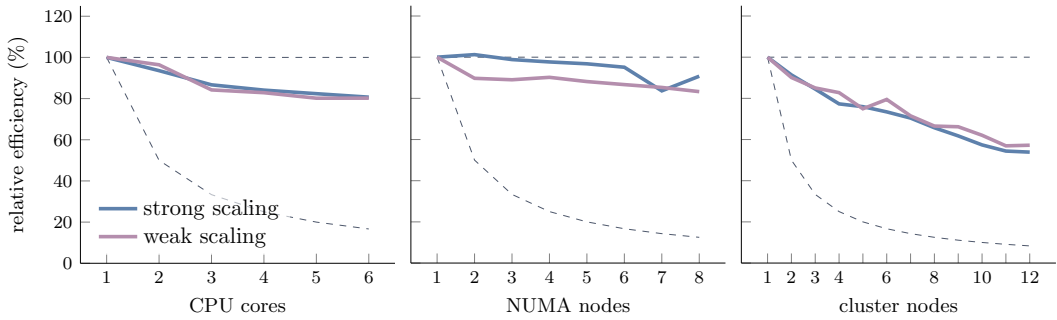
Table 4.4: Scaling efficiencies of strong and weak scaling experiments when using the maximum number of workers.

In general, the strong and weak scaling experiments show good scalability (Figure 4.8 and Table 4.4). In most cases the efficiencies are around 80% or higher. When scaling over cluster nodes the efficiencies are lower, especially in the case of the experiments with the focal operation and the example model. But even when using 12 cluster nodes it is still useful to use additional nodes to obtain model results faster, or to simulate larger problems. The strong scaling experiment of the zonal operation over NUMA nodes, shows supra-linear scaling. This implies that the performance of the zonal operation when using multiple NUMA nodes is better than can be expected given the performance when using a single NUMA node. One reason for this may be that the partition size used in each scaling experiment is determined using the problem size and maximum number of workers as used in the strong scaling experiments (Section 4.4). It is possible that this partition size is less optimal when running the model on a single NUMA node. This would then increase the latency of running the zonal operation model on a single NUMA node, and increase the associated scaling efficiencies.

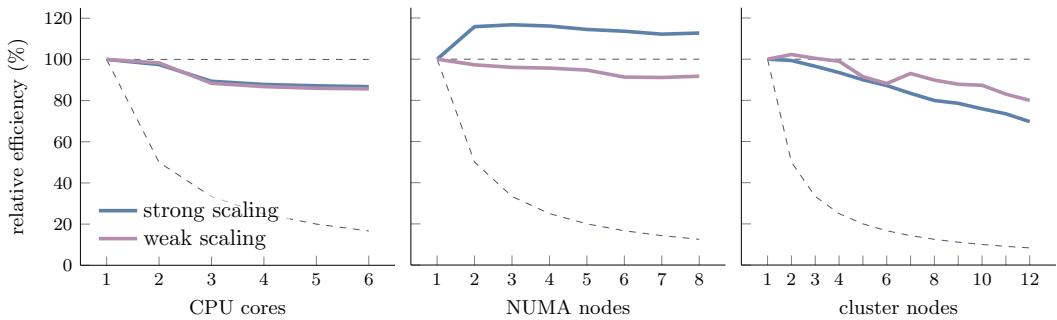
The measured throughputs (Table 4.5) show that the local operation experiment is able to provide results faster than the focal operation experiment, which is faster than the zonal operation experiment. Since the wildfire model contains more expressions per time step than the other experiments, the speed with which it is able to fill the final raster



(a) Local operation.

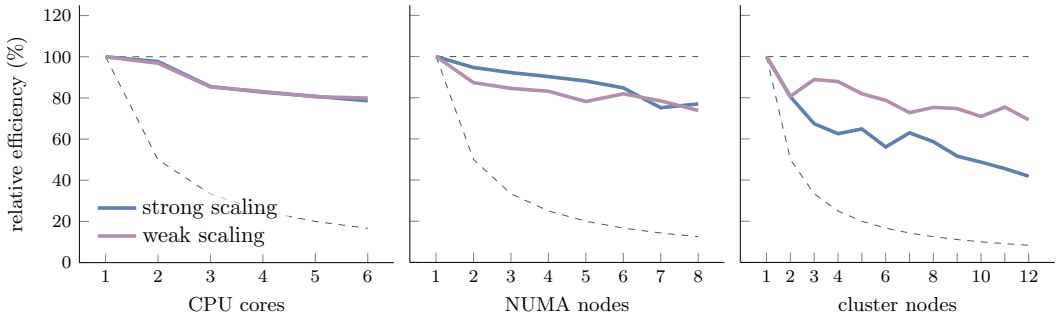


(b) Focal operation.



(c) Zonal operations.

Figure 4.8: Scaling efficiencies of experiments, over different kinds of workers. For reference, efficiencies for linear scaling (upper dashed line) and serial scaling (lower dashed line) are also shown.



(d) Example model.

Figure 4.8: Scaling efficiencies (continued).

workers	local	focal	zonal	model
6 CPU cores	0.49	0.37	0.14	0.02
8 NUMA nodes	3.37	2.54	1.04	0.14
12 cluster nodes	27.28	18.51	9.18	1.00

Table 4.5: Throughputs in MLUPS (million lattice updates per second) of weak scaling experiments when using the maximum number of workers.

at the end of each time step is lower. Given the scaling efficiencies of the experiments, throughputs increase with the number of workers.

We compared the latency of the wildfire model with the same model implemented using PCRaster¹. In an experiment using a single CPU core, with rasters of 500×500 cells and 5 000 time steps, PCRaster took 5 minutes and the new framework 5:45 minutes. These experiments used a single CPU core, and performed I/O, to different file formats.

The latencies shown in Table 4.6 show how long the experiments took. Although the amount of work per CPU core was kept more or less constant *between* experiments of different kinds of workers (see Table 4.2), the weak scaling latencies increase when going from CPU cores to NUMA nodes to cluster nodes. This is likely due to the loss in efficiency when changing the kind of worker. The latencies of the networks between NUMA nodes and cluster nodes add to the total latencies of performing an experiment.

¹ The PCRaster model is available in the source code repository associated with this chapter (Appendix A).

workers	local	focal	zonal	model
6 CPU cores	27	36	94	68
8 NUMA nodes	33	44	107	97
12 cluster nodes	55	81	163	186

Table 4.6: Latencies rounded to minutes of weak scaling experiment when using the maximum number of workers. Note that the number of time steps and the shape of the rasters used in the wildfire model experiment are smaller than the ones used in the other experiments (see Table 4.2).

4.5.2 Usability

The source code implementing the modelling framework currently contains two main layers of abstraction. The first one implements the partitioned array and related functionality for creating new arrays, distributing partitions over processes, and transporting partition data between processes. This layer uses facilities provided by the HPX library, like components for implementing array partition data servers, and serialization archives for communicating data between processes. HPX provides a relatively high level of abstraction on top of lower level abstractions, for example for managing OS threads, scheduling tasks for execution, communication between processes, and communication between cluster nodes. In the lowest abstraction level of the modelling framework code, we did not have to concern ourselves with this.

In the second layer of abstraction in the framework, the algorithms are implemented, using the functionality from the first layer. No lower level abstractions are needed when implementing algorithms. For example, in the implementation of the algorithms it is not necessary to be aware of where partitions are located, or to manage the sending and receiving of partition data. Such steps are therefore missing from the descriptions of the algorithms shown in Appendix 4.7.1.

The result of being able to separate responsibilities in multiple layers of abstraction is that it has become possible to implement the modelling algorithms using a few lines of code, especially after refactoring the code common to similar operations. For example, all code common to the binary local operations is refactored into a single C++ function. All concrete binary local operations use this function in their implementation, passing only the part that is unique for the actual operation. In the case of a parallel and distributed local operation calculating the

sum of two arrays, this part consists of a single line of code calculating the result of summing two array elements.

4.6 DISCUSSION

4.6.1 *Scalability and performance*

In this chapter we presented the design and implementation of a prototype environmental modelling framework using asynchronous many-tasks, supported by the HPX library. The initial set of local, focal and zonal operations included in our framework show good weak and strong scaling characteristics over the three kinds of workers considered. We have used a maximum number of 576 CPU cores (12 cluster nodes with 48 CPU cores each), and our results show that it is still beneficial to add more cluster nodes to be able to process more data or obtain model results faster. The scalability of the example model is comparable with the scalability of the individual operations. Apart from selecting a good partition size, the example model did not require any tuning by the model developer. The HPX runtime is able to schedule tasks generated by multiple modelling operations at the same time.

Even though the scalability we measured was good, there is still room for improvement. Also, when using our framework on larger cluster partitions, with more than the 12 nodes we considered in this study, scalability will likely continuously decrease. To improve the scalability and performance of the models created with our framework, in future research we will look into the effect of the parallel generation of tasks. Currently, the main tasks implementing the modelling operations are generated by a single process, using a single CPU core. Using more CPU cores for this, will likely increase the speed with which tasks are distributed over the processes. For the same reasons we will also research the automatic load balancing between processes. The HPX library contains facilities that make it feasible to include this in our framework (see Heller et al. [74] for an example in which this is already done). Increasingly, with the inclusion of more kinds of modelling operations, some processes may have consistently less to do than others. Because tasks follow the data, moving partitions from one process to the other automatically shifts the computational load as well. Note that such future improvements to how the framework works

internally will not influence how the models themselves are developed by the model developer, but only their scalability and performance.

We have focused here on the computational aspects of a scalable modelling framework, and disregarded that the runtime of models includes I/O as well. In practice, the time spent on I/O can be dominating the runtime of a model, and more so as the part spent on computation scales better. Scalable I/O depends on the use of parallel I/O, supported for example by MPI-IO [123] and higher level APIs using it, like NetCDF [168], HDF5 [162], or the LUE physical data model [87]. The latter API is part of the same software library as the framework described in this chapter. Investigating how to incorporate scalable I/O in the framework is an important next step.

Comparing our scalability and performance results with the results of related studies is difficult. This is because we focus on the scalability of the compute part of models, on the use of different kinds of workers, including cluster nodes, and because other studies use different operations or models in their experiments. For example, PCML [155] does not support distributed computing, scalability results are reported for a maximum of 16 CPU cores, and these include time spent on I/O. The comparison of our example model with the same model implemented with PCRaster showed that, although these two modelling frameworks have a very different implementation, the new framework containing modelling operations that are unoptimized for absolute performance approaches PCRaster's performance. Given the scalability characteristics of the new framework, the new framework will be faster and be able to process much larger problems when additional workers are used.

4.6.2 *Usability*

Our framework allows modellers to write their models using simple imperative statements similar to existing map algebra implementations. No technical details related to parallel and distributed computing are leaked to the model development interface, as illustrated by the pseudocode models in Appendix 4.7.2. Currently, the modelling operations are available as C++ API functions. Models are regular executables that can be run from the command line, either distributed or non-distributed. When run distributed, multiple copies of a model must be started. This is handled by MPI-related tools or a batch scheduler, which are available on every standard computer cluster. When

run non-distributed, there is no need for a dependency of the model on MPI, and models can be run on regular desktop or laptop computers.

When executing models, the modeller must pass a partition size to the model which results in the best performance. Having to determine this partition size is something we would like to relieve the modeller of, possibly by an automatic procedure. The integration of the APEX performance environment for runtime adaptation [81] would allow for the automatic selection of best partition sizes, for example.

Most of the envisioned target users are not C++ developers, but are familiar with the Python language. As a proof of concept, we developed a Python package containing language bindings for two local operations (available in the source code repository, Appendix A). Eventually a Python package will be made available containing all modelling operations. Note that the syntax for implementing environmental models is almost the same when developing models in C++ or Python.

We have implemented and performed experiments with an example model simulating wildfire. This model was selected because it combines only local, focal, and zonal operations. We are confident that other models, in which the same operations are combined, will also result in good scaling efficiencies. A favourable property of using asynchronous many-tasks in the implementation of modelling operations, is that it becomes less important which operations are used and in which order they are called. Compared to approaches using the synchronous fork-join paradigm, there is an increased chance of the runtime being able to schedule tasks that are ready to run on workers.

Being able to scale models over multiple nodes in a cluster has the advantage of being able to execute models faster, but also to execute larger models. As mentioned in Section 4.4, the sizes of the rasters we used were dependent on the kind of scaling experiment and the amount of memory available in a single NUMA node. In a real modelling study, the memory in all cluster nodes can be summed and used to calculate the maximum raster sizes that can be used. For example, the 12 cluster nodes used in our experiments have an aggregated amount of memory of 3 072 GiB. Assuming only rasters containing double precision floating point values and 10 state variables, similar to the wildfire model we used, this results in raster sizes of about $200\,000 \times 200\,000$ cells. Due to other software using memory, the HPX runtime using memory, and because tasks from multiple time steps can be executing at the same time, the real size will be somewhat lower. Assuming a cell size of 10 m, the example model can model wildfire for

an area of $2\,000 \times 2\,000$ km. An area the size of a quarter of the Earth's surface can be modelled when using a cell size of 100 m. Adding more nodes to the cluster partition would increase this maximum possible raster size to use for this model further.

For the model developer using our framework, the usability of the framework is important. For the framework developer, factors related to the usability of the HPX library are important, in particular, the (in)convenience of writing modelling operations in terms of asynchronous many-tasks. Although there is a learning curve involved in using asynchronous many-tasks, writing modelling operations in terms of interdependent asynchronous many-tasks is comparable to writing regular serial code. The main difference is that the framework developer cannot assume that data is available by the time the model's flow of control reaches the operation. In principle all data is referred to by futures. An operation's tasks must be defined as something that will execute once the required input data is available. The big advantage, of course, is that resulting operations scale over multiple workers. And this is achieved without the need for using explicit message passing using MPI, and the use of synchronization primitives, as needed when using OS threads. It is the responsibility of the HPX runtime to schedule tasks on workers. This supports a good software development practice of defining stacks of abstraction layers with different responsibilities, rather than mixing framework code with code unrelated to modelling.

4.6.3 *Future work*

Given the promising results of this chapter, we will continue adding more functionality and improving the existing functionality in our future research. For example, besides the topics already mentioned in this section, we will work on the integration of more advanced operations used in environmental modelling to our framework, and assess how well they, and models using them, scale. We will add operations with a higher computational load, and a less predictable spatial distribution of computational load, and less predictable dependencies between tasks, than considered in this chapter. Examples of such operations are those that operate on a hydrologic flow direction network, and operations that operate on a friction-distance path surface.

4.7 APPENDIX

4.7.1 *Algorithms*

Algorithm 1 Local operation

```
// Part 1, in input array's process:
{
  - iterate over each input array's partition
    clients, in lockstep
  - attach task to each set of input partitions
    - invoke part 2, passing input partition
      clients
  - store output partition client returned
  - return output array containing output
    partition clients
}

// Part 2, in input arrays partitions' process:
{
  - request input partitions data
  - attach task to future to input data
    - calculate output data
    - create output partition server
  - return output partition client
}
```

Algorithm 2 Focal operation

```
// Part 1, in input array's process:
{
  // Step 1: Create halo partitions around array
  - iterate over each halo partition to create
    - once halo partition's shape is known
      - create halo partition server in nearest
        partition's process
      - return halo partition client
    - store halo partition client

  // Step 2: Spawn tasks for performing focal
  // operation on central partition
  - iterate over each input array's partition
    clients
    - collect array partition clients involved in
      calculations
    - attach task to set of input partitions
      - invoke part 2, passing collection of
        input partition clients
    - store output partition client returned

  - return output array containing output
    partition clients
}

// Part 2, in input array central partition's
// process:
{
  - request input partitions data
  - attach task to future to central input
    partition data
    - calculate output data for cells whose
      neighbourhood fits within the partition
    - return output data
  - store future to output data
  - attach task to futures to bordering input
    partitions data, and output data for central
    partition
    - calculate output data for cells whose
      neighbourhood is located partly outside the
      partition
    - create output partition server
  - return output partition client
}
```

Algorithm 3 Zonal operation (continued on next page)

```
// Part 1, in input array's process:
{
  // Step 1: Per input partition, calculate a
  // statistic per zone
  - iterate over each input array's partition
    clients, in lockstep
  - attach task to each set of input partitions
    - invoke part 2, passing input partition
      clients
    - return future to partition statistics
  - store future to partition statistics

  // Step 2: Merge the zonal statistics of all
  // input partitions
  - attach task to futures to partition statistics
    - merge partition statistics
    - return array statistics
  - store future to array statistics

  // Step 3. Per input partition, translate input
  // zone to output statistic
  - iterate over each input array's partition
    clients
  - attach task to future to array statistics
    - invoke part 3, passing zone partition
      client and array statistics
    - return output partition client returned
  - store output partition client

  - return output array containing output
    partition clients
}

// Part 2, in input arrays partitions' process:
{
  // Calculate a statistic per zone
  - request input partitions data (value and zone)
  - attach task to future to input partitions data
    - calculate statistic of value per zone
    - return statistics
  - return future to statistics
}
```

Algorithm 3 Zonal operation (continued)

```
// Part 3, in input arrays partitions' process:
{
  // Translate input zone to output statistic
  - request input partition data (zone)
  - attach task to future to input partition data
    - reclass zone based on statistics
    - create output partition server
  - return output partition client
}
```

4.7.2 Scripts

In the scripts shown in this section, `clone` refers to an existing distributed partitioned array. An operation like `uniform` needs this information to be able to create an output array. The code of the actual experiments is very similar to the pseudocode shown here, but is implemented in C++. It can be found in the repository associated with this chapter (Appendix A).

When the experiment models are executed by the modelling framework, concurrent tasks are generated that execute in parallel on multiple workers, potentially on multiple cluster nodes. Note that none of the model scripts contain technical details related to parallel and distributed computing.

Listing 4.2: Pseudocode of local operation experiment.

```
# Initialize state with random numbers
# between zero and a large value
state = uniform(clone, 0, maximum)

for t in range(nr_time_steps):
  # Raise the square root of each cell
  # value in state to the power of two
  state = pow(sqrt(state), 2)
```

Listing 4.3: Pseudocode of focal operation experiment.

```

# Initialize state with random numbers
# between zero and a large value
state = uniform(clone, 0, maximum)

# A 3x3 boolean kernel filled with true
# (== kernel weight of 1)
kernel = box_kernel(1, true)

for t in range(nr_time_steps):
    # For each cell in state, calculate
    # the mean value of the cells that
    # lie within the kernel
    state = focal_mean(state, kernel)

```

Listing 4.4: Pseudocode of zonal operation experiment.

```

# Initialize state with random numbers
state = uniform(clone, -1e6, 1e6)

for t in range(nr_time_steps):
    # Randomly redistribute 101 classes
    zones = uniform(clone, 0, 100)

    # Per zone in zones, calculate the
    # sum of state values and return these
    # sums
    state = zonal_sum(state, zones)

```


SCALABILITY AND COMPOSABILITY OF FLOW ACCUMULATION ALGORITHMS BASED ON ASYNCHRONOUS MANY-TASKS

Adapted from: K. de Jong, D. Panja, D. Karssenberg, and M. van Kreveld. "Scalability and composability of flow accumulation algorithms based on asynchronous many-tasks." In: *Computers & Geosciences* 162 (2022), p. 105083. DOI: 10.1016/j.cageo.2022.105083

Models simulating the state of the biological and physical environment can be built using frameworks that contain pre-developed data structures and operations. To achieve good model performance it is important that individual modelling operations perform and scale well. Flow accumulation operations that support the use of criteria for selecting how much material flows downstream are an important part in several Earth surface simulation models. For these operations, no algorithms exist that perform, scale, and compose well. The objective of this study is to develop these algorithms, and evaluate their performance, scalability, and composability. We base our algorithms on the asynchronous many-task approach for parallel and concurrent computations, which avoids the use of synchronization points and supports composability of modelling operations. The relative strong and weak scaling efficiencies when scaling a flow accumulation operation over six CPU cores in a NUMA node are 83% and 84% respectively. The relative strong and weak scaling efficiencies when scaling a case-study model over four cluster nodes are 73% and 84%. Our algorithms are composable: the latency of executing two flow accumulation operations combined is lower than the sum of their individual latencies.

5.1 INTRODUCTION

The changing state of the biophysical environment through time and space can be simulated using computer models. Modelling frame-

works contain data structures and operations which can be used to develop simulation models in less time, by model developers who do not have to know about the details involved in implementing the data structures and operations. Given the continuous increase in temporal and spatial extent and resolution of datasets, and the subsequent increase of model complexity to incorporate more detailed environmental process descriptions, it is important that modelling frameworks support the development of models that perform and scale well over additional hardware. For some modelling operations good performance and scalability is easier to achieve than others. An important aspect of a modelling operation that potentially limits its performance and scalability when parallelized is the spatial dependency of output values on input values. Parallelizing modelling operations generally involves dividing the spatial domain into partitions, each of which is processed by a separate worker, like an OS thread on a CPU core. In case of spatial dependencies of output values on input values, data must be exchanged between workers. The performance and scalability of such an operation depends on how well workers are able to cooperatively carry out the total amount of work. Examples of modelling operations with spatial dependencies of output values on input values are spreading operations and flow routing operations [24].

The current chapter concerns routing of material over a D8 flow direction raster using flow accumulation operations. In a D8 flow direction raster, each cell is assigned a direction of one of its 8 neighbours to which it drains [128]. This results in a dense non-divergent directed acyclic graph of which the main branches correspond with the hydrologic network of streams and rivers. Flow accumulation operations are part of several Earth surface simulation models, examples of which are LISFLOOD [23], used for the European Flood Awareness System (EFAS), and the PCR-GLOBWB global water balance model [160].

Flow accumulation algorithms that can be found in the literature solve the problem of transporting *all* material in downstream direction [9, 35, 107, 135, 157, 179], but sometimes flow accumulation operations are required that use a criterion to split the total amount of material entering a cell (inflow) into an amount that is transported downstream (outflow) and an amount that remains in the cell (residue, Table 5.1, [98]). An example of such a criterion is a threshold representing the minimum amount of material that has to be received by a cell before excess material starts to be transported downstream. Using this operation, henceforth referred to as `accu_threshold`, the process of

Operation	Outflow calculation
<code>o = accu(d, i)</code>	The total amount of inflow material.
<code>o, r = accu_fraction(d, i, f)</code>	A fraction of the amount of inflow material.
<code>o, r = accu_threshold(d, i, t)</code>	The amount of inflow material exceeding a threshold.
<code>o, r = accu_capacity(d, i, t)</code>	The amount of inflow material below a threshold.
<code>o, r = accu_trigger(d, i, t)</code>	The total amount of inflow material, once a threshold is exceeded.

Table 5.1: Examples of flow accumulation operations and criteria used for calculating the amount of outflow material *o* and residue *r* per cell, given the flow directions *d*, total amount of inflow of material *i*, and either a fraction *f* or a threshold *t*. All arguments are rasters.

Hortonian overland flow [76] can be simulated, for example. Examples of other processes that can be simulated by these operations are loss of material while on transport (using `accu_fraction`), flow through a sewage system (using `accu_capacity`), and mass movements (using `accu_trigger`).

The various existing flow accumulation algorithms have different computational properties. We focus on flow accumulation algorithms that use CPU cores rather than GPU devices. Distributing work over multiple GPUs in multiple cluster nodes complicates the algorithms and makes the implementation less portable. In Zhou et al. [179] a review of serial algorithms is provided and a new algorithm is presented that offers better performance than those reviewed. Although this algorithm only considers the basic flow accumulation function, without using a criterion, it can be extended to support the flow accumulation operations that do use one. One limitation of the algorithm is that it is not capable of using multiple CPU cores, which limits its applicability to relatively small problems. In Kotyra et al. [107] it has been concluded that a parallel version of the algorithm by Zhou et al. [179] performs best compared to other parallel algorithms they tested; one limitation of their algorithm is that it is not capable of using multiple nodes in a computer cluster. Barnes [9] presents an approach for distributing flow accumulation computations over multiple processes. For this algorithm to work, the spatial domain is partitioned into rectangular partitions. Like in the case of the algorithm presented in Zhou et al. [179], this algorithm only considers flow accumulation without using a criterion, but this algorithm cannot be easily extended

to support the other kinds of flow accumulation operations. The algorithm by Barnes [9] requires that there is a linear relation between the amount of material entering a partition and the amount leaving it. This allows the algorithm to calculate a final result efficiently, in a single concurrent step per partition, without having to iterate over partitions containing upstream parts of large scale streams to partitions containing downstream parts. Given our requirement of being able to use a criterion, we cannot use the final steps of this algorithm. The criteria used by `accu_threshold`, `accu_capacity`, and `accu_trigger` require that the total amount of inflow of material in a cell is known, before the amounts of residue and outflow of material from that cell can be calculated.

The fact that, in the general case, flow accumulation results for cells of streams that flow from spatial domain partition to domain partition must be calculated in order, going from upstream to downstream direction, implies that there is a temporal load imbalance between partitions. The larger the flow direction raster and the more partitions involved in calculating the flow accumulation result, the larger this load imbalance can become. This is important when flow accumulation is used in a calculation involving other operations as well, like in a simulation model or a GIS workflow. Performance and scalability of such calculations will be limited when subsequent calculations have to wait on the last partition of the flow accumulation operation to finish. In case of such a synchronization point, workers like CPU cores or even whole cluster nodes may be drained of useful work to do. Ideally, partitions for which flow accumulation operation calculations have finished should already participate in calculations of other operations.

We call a set of modelling operations *composable* when the time it takes the set to finish executing is shorter than the sum of their individual latencies. To the best of our knowledge no existing flow accumulation algorithm has been designed taking into account that the flow accumulation operation will be combined with other operations.

The problem we try to solve is the parallelization and distribution of a set of flow accumulation algorithms, some of which use a criterion for determining how much material flows downstream from each cell. As an additional requirement, we want the resulting operations to be composable with other operations. Our objective, therefore, is to design a general scheme for flow accumulation algorithms that enables them to perform well, scale well, and compose well with other operations.

To reach our objective, we make use of an approach for writing parallel and distributed software called asynchronous many-tasks (AMT), as implemented in the HPX C++ library for parallelism and concurrency [91]. One advantage of using AMT is that it allows the software developer to define tasks, representing an amount of work to be performed, to be asynchronously scheduled, potentially allowing work from multiple operations to be scheduled concurrently and executed in parallel. We designed and expressed our new algorithms in terms of AMT concepts and the HPX API, added prototype implementations to the LUE modelling framework [88], and performed experiments to assess their strong and weak scalability, and their composability.

Our results show that our AMT-based algorithms are capable of using additional hardware efficiently, and perform well when combined. The strong and weak scaling efficiencies when scaling a flow accumulation operation of six physical CPU cores in a NUMA node are 83% and 84% respectively. When scaling a case-study model over four cluster nodes containing 48 physical CPU cores each, the strong and weak scaling efficiencies are 73% and 84% respectively. Also, the new algorithms are composable: the latency of executing two flow accumulation operations combined is lower than the sum of their individual latencies.

The organization of this chapter is as follows. We start with an introduction of AMT, HPX and the LUE modelling framework (Section 5.2). We then describe our flow accumulation algorithms (Section 5.3), and the experiments we performed (Section 5.4). The results of the experiments can be found in Section 5.5. We finish the chapter with a discussion of the results and our conclusions (Section 5.6).

5.2 AMT, HPX, AND THE LUE MODELLING FRAMEWORK

We start with a brief introduction of some major aspects of AMT, the HPX implementation thereof [91], and the use of AMT and HPX in the LUE environmental modelling framework [88].

With the AMT programming model the software developer defines relatively small tasks of work that need to be performed, and the dependencies between them. Once tasks have been created, the AMT runtime system is responsible for executing them in a correct order, using the available workers (e.g. CPU cores). To increase the chance that an AMT program performs and scales well, it should create enough tasks that are ready to run to keep all workers busy. Tasks are therefore

spawned asynchronously, and they must have as few dependencies as possible between them.

HPX is an implementation of the AMT programming model and run-time. It is an open source software library written in portable C++11/14/17/20 code. With HPX, every system, ranging from laptops to computer clusters, is represented as a single abstract machine, containing one or more localities. For our purposes, localities are equal to operating system processes, so we will use the more familiar term process. Each process exposes plain actions and component actions. Plain actions are globally accessible free functions without state, and component actions are globally accessible member functions of objects with state. The software developer uses the HPX API to define these actions. An HPX task is a lightweight HPX thread. A task can call an action and can execute locally or remotely, in a different process. When an HPX task is spawned asynchronously, a future object to the result is returned immediately. This object represents a result that may not be computed yet, and it allows one or more continuations to be attached, which get called once the future they are attached to becomes ready. Futures can be composed to represent relations between tasks. HPX components are globally addressable (using an ID) instances of classes. A component server is the actual instance, located in a process. A component client is a lightweight object providing access to a possibly remote server instance. It is semantically equal to a shared future to the ID of the remote server instance. HPX channel components allow asynchronous communication between different tasks in different processes.

LUE is a modelling framework targeted at domain experts, like hydrologists, soil scientists and biologists. The modelling operations are inspired by map algebra [166]. LUE currently contains a set of local, focal, zonal, and global operations. In LUE models, time is typically discretized in time steps, and space in raster cells. For model developers LUE provides a Python language binding, which allows them to use the common procedural programming paradigm to implement models. Models run unchanged on laptops and computer clusters. To the modeller, LUE models look similar to models created with other map algebra implementations, like ArcGIS [46], GDAL [58], GRASS [125], and PCRaster [44]. The core data structure used in the current LUE API is the partitioned array. A partitioned array contains array partition clients referring to partition servers containing a rectangular section of the overall array. In the implementation, LUE modelling

operations attach continuations to array partition clients. These continuations asynchronously spawn work and immediately return new array partition clients, to be used as input for other operations. Depending on the dependencies between the array partition clients, tasks from multiple modelling operations can be scheduled for execution at the same time, in parallel. LUE distributes array partition servers, containing the array data, evenly over the processes. Work generated by modelling operations translate input partitions to output partitions, and execute in the same processes as the partitions they operate on. In case of local and focal operations, an even distribution of partitions over processes results in an even distribution of computational load.

5.3 FLOW ACCUMULATION

Our flow accumulation algorithms combine and extend the efficient algorithm of Zhou et al. [179] and the distributed algorithm of Barnes [9]. In this section we describe our algorithms and show how we applied AMT. We use the `accu_threshold` operation as an example. The other flow accumulation algorithms use the same approach. A call to this operation looks like this:

```
outflow, residue = accu_threshold(flow_direction, material, threshold)
```

Like the flow direction, the material and threshold arguments vary through space, and are represented by arrays.

5.3.1 Overview

Our algorithm works with partitioned arrays (Section 5.2). Cells in a partitioned flow direction array can be classified according to their location within the flow direction graph and within a partition (Figures 5.1 and 5.2). Cells that only receive material from upstream cells that are located in the same partition are called intra-partition stream cells. Cells that receive material from at least one upstream cell that is located in another partition are called inter-partition stream cells. A partition output cell provides material for a cell in a neighbouring partition.

Per partition, flow accumulation calculations start with intra-partition stream cells and continue with inter-partition stream cells. Within the intra-partition stream cells, calculations start at ridge cells, which do not receive input from another cell, and terminate at inter-partition

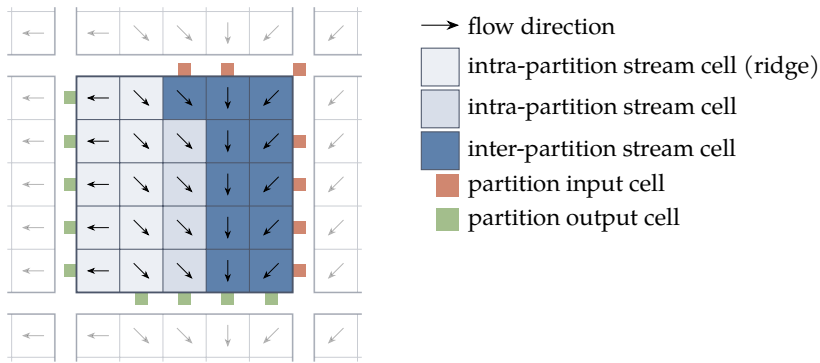


Figure 5.1: Classification of cells in a single partition of a flow direction array. Partition output cells are marked with green “sockets”, directed at the relevant neighbouring partition, and partition input cells are marked with orange sockets.

stream cells, sinks, or partition output cells. Within the inter-partition stream cells, calculations start at partition input cells and stop at sinks, or partition output cells.

A flow accumulation calculation for a cell starts with adding the external material—passed in as an argument to the operation—to the amount of inflow material that the cell received from upstream, if any. Based on the threshold criterion also passed in, the total amount of material in the cell is then split into an amount of residue and an amount of outflow material.

In order to be able to visit all cells in the correct order, going from upstream to downstream in the flow direction graph, we first calculate the number of directly neighbouring cells that drain into each cell. In Zhou et al. [179] this is called the number of input drainage paths (NIDP). Cells with an NIDP of zero do not receive material from any neighbour. Most of these cells are ridge cells, but some may be positioned at the border of the raster and part of a large scale stream flowing into the area represented by the raster. For the purpose of our algorithm, this latter kind of cells can be treated as ridge cells. Cells with an NIDP of eight must be sink cells. Cells with an NIDP between one and seven are junction cells, some of which may be sink cells—surrounded by at least one no-data cell—but the majority will drain to a downstream cell.

Given the NIDP values of each cell, per partition, ridge cells can be found and used as starting points for flow accumulation calculations. Once calculations for a ridge cell have finished and assuming it has a

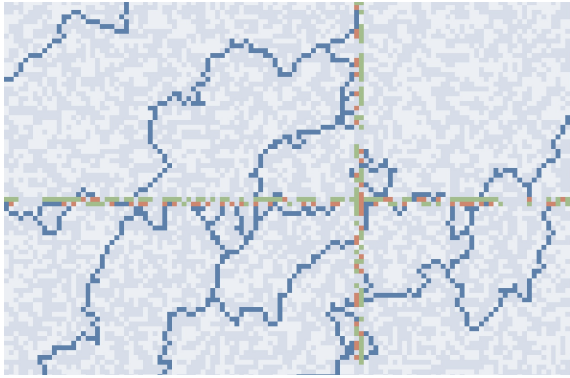


Figure 5.2: Flow directions reclassified according to the classes from Figure 5.1, for a small area of the MERIT Hydro dataset [176]. The map shows the borders of four adjacent partitions. Most cells at the borders are partition output cells, producing material to forward to matching partition input cells of a neighbouring partition. Relatively few cells are part of an inter-partition stream; most are intra-partition stream cells.

downstream cell, the resulting outflow is added to the material of the downstream cell and its NIDP value is decreased by one. If the updated NIDP value of the downstream cell has become zero, the current cell is the last cell draining into it. In that case, the flow accumulation procedure is repeated for that cell. The procedure terminates when a downstream cell is encountered which is either a junction cell with an updated NIDP value that is larger than zero, a sink cell, or a partition output cell. Once all ridge cells in a partition have been used as starting points this way, flow accumulation calculations are finished for all intra-partition stream cells. All material “produced” by these cells has been “deposited” in inter-partition stream cells, sink cells, and partition output cells.

Next, for each partition, the input cells for which material is available in the corresponding partition output cells in the neighbouring partition(s) are used as starting points for the same flow accumulation procedure as used during the calculations for the intra-partition stream cells. Once calculations for all input cells and the inter-partition stream cells downstream of them have finished, the flow accumulation calculations for the partition have finished.

Concluding, our algorithm performs these three steps for each partition: 1) calculate the NIDP for each cell, 2) calculate the flow accumulation results for the intra-partition stream cells, and 3) calculate the flow accumulation results for the inter-partition stream cells.

Compared to the algorithm by Zhou et al. [179], we have split the calculations in two steps: one to solve the flow accumulation for intra-partition stream cells, and one for solving the flow accumulation for inter-partition stream cells. This is necessary since we use a partitioned array, and in general, partitions contain inter-partition stream cells that can only be calculated once the flow accumulation calculations for all upstream cells have finished. Note that in general, partitions cannot be ordered according to their position along an inter-partition stream. A large scale stream may visit the same partition multiple times and multiple large scale streams may pass through the same partition (Figure 5.2).

Compared to the algorithm by Barnes [9], we have changed the procedure for calculating the results for the inter-partition stream cells. As described in Section 5.1, this cannot be done in a single concurrent step, but requires multiple steps, propagating material from partition input cells and through inter-partition stream cells as the material becomes available from upstream partition output cells.

5.3.2 *Parallelization*

A number of concurrent aspects can be identified in the above procedure. First, the NIDP values can be calculated in parallel for each partition. A small amount of information about which partition output cells flows into which partition input cells must be communicated. Second, the flow accumulation results for intra-partition stream cells can be calculated in parallel for each partition. Information about material reaching partition output cells must be communicated to allow this material to be used as input for partition input cells in a subsequent step. Third, propagating material from a partition input cell through a partition can be done in parallel for each partition.

5.3.3 *Application of AMT*

The next list shows the steps of our flow accumulation algorithm in terms of the AMT approach.

1. Create channels for exchanging information about partitions between tasks. Each channel server instance is instantiated in the process of the partition for which information is sent.

2. For each partition, asynchronously spawn a task to calculate the results of the flow accumulation calculations. Each of the tasks performs these steps:
 - a) Asynchronously spawn a task that calculates the NIDP for each cell. Use channels to send information about which partition input cell in a neighbouring partition receives material from this partition, to a task monitoring the relevant channel for this neighbouring partition.
 - b) Asynchronously spawn a task that calculates the flow accumulation results for all intra-partition stream cells. Use channels to send information about material flowing into a partition input cell in a neighbouring partition, to a task monitoring the relevant channel for this neighbouring partition.
 - c) Asynchronously spawn a task that calculates the flow accumulation results for all inter-partition stream cells. Again, use channels to send information about material flowing into a partition input cell in a neighbouring partition, to a task monitoring the relevant channel for this neighbouring partition.
3. Return partitioned arrays for outflow and residue. Note that these arrays are returned before the flow accumulation calculations have finished. They may not even have started yet.

All tasks are spawned asynchronously and return futures to results immediately. Each next task depends on the results of the previous task(s) and will only be created and scheduled for execution by the HPX runtime after these results have become available. Within each algorithmic step, tasks performing work for a certain partition only depend on tasks performing work for directly neighbouring partitions. These dependencies are represented by the channels which are used to exchange information.

When calculating the results for the inter-partition stream cells in a partition, work is only performed once material becomes available for a partition input cell. Until that is the case, the task is automatically suspended by the HPX runtime. It is important to note that the topology of the flow direction graph is not explicitly used to order tasks according to their relative position along the inter-partition streams. Once a task calculating the flow accumulation results for a partition

has received and propagated material for all its partition input cells, the work for the partition is done. As soon as this happens, the corresponding outflow and residue result partition clients are marked as ready, and these partitions can be used in subsequent modelling operations.

5.3.4 *Algorithm details*

All processes are worker processes that are involved in the computations. Array partition server instances are evenly distributed over all processes and work associated with partitions is distributed accordingly. Partitioned arrays, containing the partition client instances are located in the root process, which is one of the worker processes. In this process, all modelling operations are executed, which create work for all worker processes to execute. In the case of `accu_threshold`, the following steps are performed in this process:

1. Create an array with for each partition a communicator object, containing two sets of at most eight channel objects. One set is used to send information about which partition output cell flows into which partition input cell in one of the eight neighbouring partitions. The other set is used to receive this information about the eight neighbouring partitions.
2. Create a similar communicator array, but with channels for exchanging information about an amount of material a partition input cell receives.
3. For each partition:
 - a) Asynchronously call a global action in the worker process of the partition that will perform all required flow accumulation calculations for that partition. Pass in the partition clients for flow direction, external inflow of material, and the threshold, and a communicator for NIDP and one for material. This call immediately returns partition client instances for outflow and residue.
 - b) Store returned partition client instances in their respective output partitioned arrays.

These are the main steps performed by the global action in the worker process:

1. Once the flow direction partition is ready:
 - a) Send locations of partition input cells in neighbouring partitions that will receive material from the current partition using the corresponding channel in the NIDP communicator. Per neighbouring partition direction (north, north-east, east, south-east, south, south-west, west, north-west), store the location of the partition output cells.
 - b) Asynchronously receive locations of partition input cells that will receive material from neighbouring partitions. Per neighbouring partition direction, store the location of these partition input cells.
 - c) Once the collections of partition input cells have been received, calculate the NIDP for all cells in the partition.
2. Once the flow direction partition, external inflow material partition, threshold partition, NIDP partition and collections of partition output cells are ready:
 - a) For each cell with an NIDP of zero:
 - i. Start the flow accumulation procedure until a junction, sink, or partition output cell is reached.
 - ii. If a partition output cell is reached, send location and outflow to the input cell in a neighbouring partition using the associated channel in the material communicator. If this was the last output cell associated with the neighbouring partition, close the channel.
 - b) Immediately return futures to the outflow partition data, residue partition data, and the updated NIDP counts
3. Once results of the intra-partition stream cell calculations are ready:
 - a) Asynchronously create tasks that will each monitor a channel for incoming material sent from tasks handling neighbouring partitions. Once material is received:
 - i. Start the flow accumulation procedure until a junction, sink, or partition output cell is reached.
 - ii. If a partition output cell is reached, send location and outflow to the input cell in a neighbouring partition using the associated channel in the material communicator.

CPU	2 AMD EPYC 7451 (2 packages)
NUMA nodes	8 (4 / package)
Cores	48 (6 / NUMA node)
Clock frequency	2.3 GHz
L1d / L1i	32 / 64 KiB / core
L2	513 KiB / core
L3	8 192 KiB / 3 cores
RAM	256 GiB (32 GiB / NUMA node)
OS	CentOS 7
GNU GCC	version 10.3.0
HPX	version 1.7.1 [93]
MPI	Open MPI version 4.0.5

Table 5.2: Hardware and software platform of cluster nodes used in experiments. All cluster nodes are interconnected with InfiniBand.

iii. Stop when all partition input cells have received a value.

Note that only the final step, when material for partition input cells is received from tasks managing the eight neighbouring partitions, requires the use of a mutex to serialize concurrent access to the NIDP, outflow, and residue partitions. In the rest of the algorithm, these partitions are only written to by a single task.

5.4 EXPERIMENTS

We performed various experiments to characterise the performance, scalability and composability of our new flow accumulation algorithms. In all experiments, we used the MERIT Hydro dataset [176] for the African continent. This dataset has a 3 arc-second resolution, which corresponds to almost 90 m resolution at the equator. It contains $87\,600 \times 84\,000$ raster cells and represents a realistic high resolution dataset that can be used in global and continental scale modelling studies. All experiments were performed on one or more equivalent cluster nodes (Table 5.2). Even though the latency of simulation models is a combination of time spent on computing output values and on I/O, in our experiments we only considered the time spent on the compute part.

5.4.1 *Algorithm*

The algorithms described in Section 5.3 asynchronously spawn various kinds of tasks with dependencies between them. To gain insights into when these tasks get scheduled at runtime, we generated a trace for a single run of two flow accumulation operations, using the same script as used in the composability experiment (Section 5.4.3). We performed the experiment on the same dataset as used in the weak scaling experiment over CPU cores of `accu_threshold` (Table 5.3) when using 6 CPU cores. This array has $30\,000 \times 30\,000$ cells and contains relatively few no-data cells. The results of this experiment are given and analysed in Section 5.5.1.

5.4.2 *Performance and scalability*

To put the results of the scalability experiments into perspective, we compared the performance of our new operations with the performance of similar operations from the PCRaster environmental modelling framework [101]. We compared the latencies of a single `accu` and a single `accu_threshold` call for the southern half of the African continent ($56\,059 \times 44\,956$ raster cells). The experiments were performed on a single CPU core. All variables (inflow, threshold, outflow, and residue) were represented by arrays containing 32 bit floating point elements.

We performed scalability experiments, on individual calls to a flow accumulation operation, and on a case-study model in which a flow accumulation operation was combined with several local operations, with data dependencies between them. The relative fraction of local operations versus flow accumulation operations used in the case-study model is comparable to existing hydrological models in which flow accumulation is used, like the PyCatch catchment model [110] and the PCR-GLOBWB global water balance model [160]. In the case-study model a call to `accu_threshold` is surrounded by 57 local operations. A feedback variable is used to add a data dependency between operations from consecutive time steps.

To characterise the ability of each computation to use additional workers to perform work faster, we calculated the relative strong scaling efficiencies (RSE_{strong}). These are calculated by dividing the latency $T_{S,1}$ on a single worker by the latency $T_{S,P}$ on P workers, multiplied by P , while the problem size is kept constant (Equation 5.1).

To characterise the ability of each model to use additional workers to perform more work, we calculated the relative weak scaling efficiencies (RSE_{weak}). These are calculated by dividing the latency $T_{W,1}$ on a single worker by the latency $T_{W,P}$ on P workers, while the problem size scales according to the number of workers (Equation 5.2).

$$RSE_{strong} = \frac{T_{S,1}}{P \times T_{S,P}} \times 100\% \quad (5.1)$$

$$RSE_{weak} = \frac{T_{W,1}}{T_{W,P}} \times 100\% \quad (5.2)$$

To be able to use the differences between kinds of workers in the interpretation of the results of the scalability results, we performed the experiments over three kinds of workers: 1) the 6 CPU cores within a single NUMA node, 2) the 8 NUMA nodes within a single cluster node, and 3) 4 cluster nodes within a cluster partition.

When selecting the size of the dataset to use in scalability experiments we took two factors into account. It had to be large enough to result in latencies that hide normal fluctuations in latencies due to the scheduling of the processes by the OS. But it had to be small enough to fit in the memory of a single CPU worker, and to result in latencies small enough to be feasible. Because of this, it was not possible to perform scalability experiments over NUMA nodes and cluster nodes for a single flow accumulation operation with the original MERIT Africa dataset. This dataset does not contain a large enough subset without a large number of no-data cells in them. We therefore resampled the original dataset to double its resolution. This allowed us to perform scalability experiments over NUMA nodes, but still not over cluster nodes. We did perform scalability experiments over all kinds of workers for the case-study model. Since this model contains more operations, the latencies are higher, and the subset to use could be smaller. The subsets of the MERIT Africa dataset and the resampled version thereof used in the scalability experiments are listed in Table 5.3. Figure 5.3 shows the bounding boxes used for the weak scalability experiments.

Since tasks are created in continuations attached to array partitions, the size of these partitions determines how many tasks will be created during the execution of a computation. Small partitions result in many tasks, increasing the chance that all workers always have enough work to do, but also increasing the overheads of managing these tasks. Large

	worker	dataset	array size
accu_	CPU core	MH1	12 000 × 12 000
threshold	NUMA node	MH2	30 000 × 30 000
model	CPU core	MH1	4 000 × 4 000
	NUMA node	MH1	10 000 × 10 000
	cluster node	MH2	28 000 × 28 000

Table 5.3: The scalability experiments performed for two kinds of jobs: a single call to a flow accumulation operation, and a model in which the flow accumulation operation is combined with local operations. Two kinds of datasets are used: the original MERIT Hydro dataset for the African continent (MH1) and a resampled version thereof, with a twice as high resolution (MH2). Array sizes correspond to a subset of the dataset centered around a cell in the middle of the continent (Figure 5.3). In case of weak scalability, the array sizes shown are scaled with the number of workers.

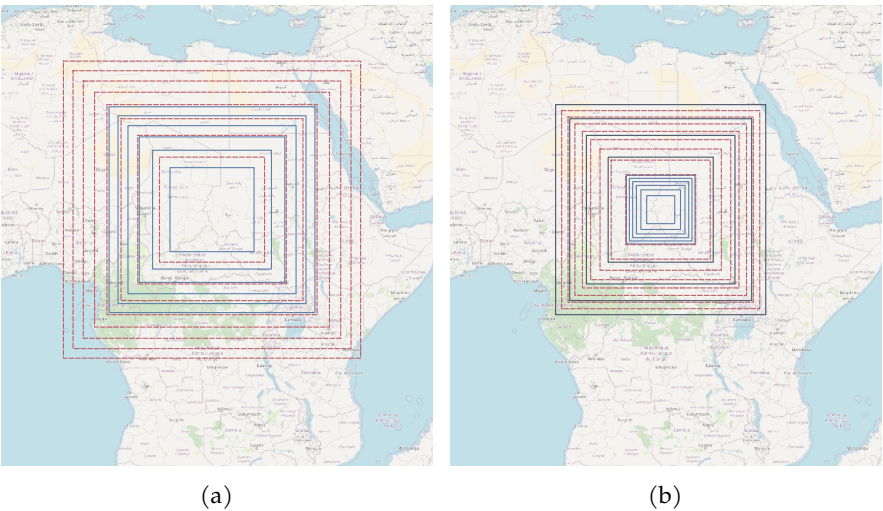


Figure 5.3: Bounding boxes of arrays used for weak scalability experiments for accu_threshold (a) and the case-study model (b). In solid blue the 6 areas used for scaling over the CPU cores within a NUMA node. In dashed red the 8 areas used for scaling over the NUMA nodes within a cluster node. In solid gray the 4 areas used for scaling over the cluster nodes within a cluster partition. (Base map and data from OpenStreetMap and OpenStreetMap Foundation.)

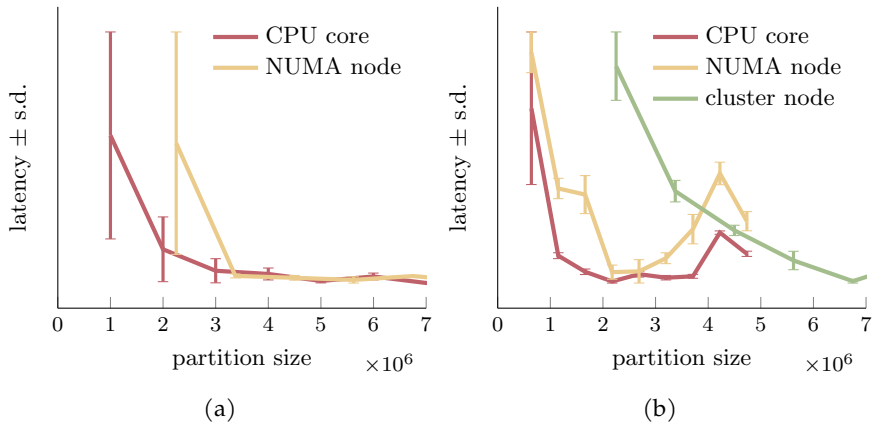


Figure 5.4: For the scaling experiment for `accu_threshold` (a) and the case-study model (b), the latencies of running the experiment for a range of partition sizes (number of cells) on the maximum number of workers and maximum array size. The actual latencies are different for the different kinds of workers. The y-axis starts at zero and increases linearly.

partitions result in few tasks, decreasing the task scheduling overheads, but also decreasing the chance that workers will always have useful work to do. For each combination of computation and a kind of worker there is a range of good partition sizes for which the computation performs best. Before each scalability experiment, we determined a good partition size to use. For this we ran each computation on the maximum number of workers used in the particular scaling experiment with the maximum array size used in the weak scaling experiment. To be able to determine the variability in the latencies, we performed each partition size experiment 5 times.

The partition sizes we used in the scalability experiments are listed in Table 5.4 and based on the distribution of latencies over various partition sizes (Figure 5.4). For appropriate partition sizes, associated with small latencies, the variability in latencies is low. We therefore did not run the strong and weak scalability experiments multiple times.

5.4.3 Composability

In order to characterise the composability of the flow accumulation operations, we used a model containing two calls to a flow accumulation operation (Listing 5.1). In an actual model, the first one could simulate water transport, while the second one uses the outflow result

	worker	partition size (no. of cells)
accu_	CPU core	$2\,500 \times 2\,500$ (6.25×10^6)
threshold	NUMA node	$2\,500 \times 2\,500$ (6.25×10^6)
model	CPU core	$1\,500 \times 1\,500$ (2.25×10^6)
	NUMA node	$1\,500 \times 1\,500$ (2.25×10^6)
	cluster node	$2\,500 \times 2\,500$ (6.25×10^6)

Table 5.4: Partition sizes used in scaling experiments.

for simulating the transport of sediment. The reason we used two calls to the same flow accumulation operation is that we relate the differences in latencies between model runs to the total latency of the model runs. Using two operations with very different latencies makes the results more difficult to interpret. We compared the latencies of executing the model with and without a synchronization point between the operations. The synchronization point, represented by a call to `wait_all` in Listing 5.1, prevents the execution of the second operation until all output partitions of the first operation are ready. Without a synchronization point, the second operation is executed as soon as the first one has finished attaching continuations to its input partitions. The hypothesis is that, in case of load imbalance, composable operations result in lower model latencies by preventing workers from being drained of work. We performed the experiments on the same dataset as used in the weak scaling experiment over NUMA nodes of `accu_threshold` (Table 5.3) when using 8 NUMA nodes. This array has $85\,000 \times 85\,000$ cells and contains relatively few no-data cells. We executed each model variant 10 times and selected the smallest latencies.

Listing 5.1: Model used in composability experiment.

```
# Arguments to flow accumulation are ready
# when operation is called
outflow, residue = accu_threshold(
    flow_direction, material, threshold)

if synchronize:
    wait_all([outflow, residue])

# Use output of flow accumulation
outflow, residue = accu_threshold(
    flow_direction, outflow, threshold)

wait_all([outflow, residue])
```

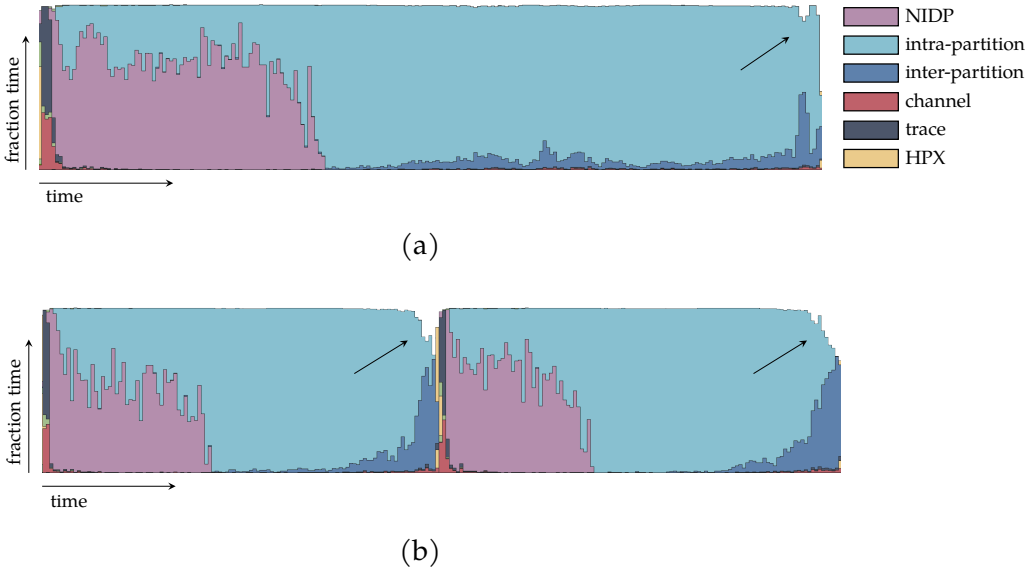


Figure 5.5: Traces of flow accumulation tasks created by two calls to `accu_threshold` (Listing 5.1): one without a synchronization point (a) and one with a synchronization point (b). Both traces show the last part of the full trace (≈ 17.5 s), excluding time spent on initializing the runtime and reading input data. The arrows identify the moments the CPU cores start to be drained of useful work to do. Trace tasks are an artefact of generating the trace. HPX tasks are a detail of the runtime. The other kinds of tasks correspond with the ones described in Section 5.3.3.

5.5 RESULTS

5.5.1 *Algorithm*

The trace shows which flow accumulation tasks are executing over time (Figure 5.5a). All the time different kinds of tasks are executing in parallel. The kind of task spending the most time on the CPU cores changes over time. The sequence corresponds with the steps performed per partition by our algorithms: NIDP, intra-partition, inter-partition (Section 5.3). Concurrent tasks, operating on different partitions, get scheduled in parallel.

5.5.2 *Performance and scalability*

Calculating a result for the southern half of the African continent with our new `accu` operation took 1.1 minutes while PCRaster's version

	worker	RSE_{strong}	RSE_{weak}
accu_ threshold model	CPU core	83%	84%
	NUMA node	69%	56%
	CPU core	52%	77%
	NUMA node	48%	67%
	cluster node	73%	84%

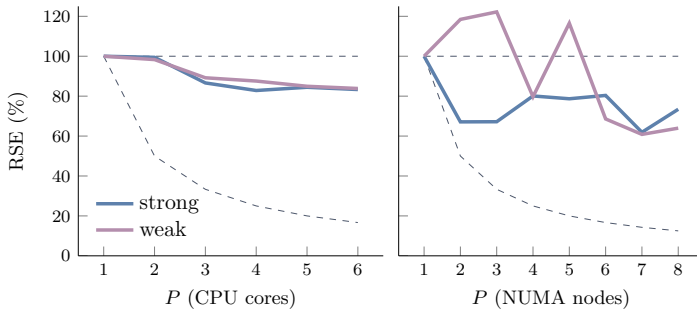
Table 5.5: Relative strong and weak scaling efficiencies.

took 3.2 minutes. Our new `accu_threshold` operation took 1.4 minutes while PCRaster’s version took 3.3 minutes.

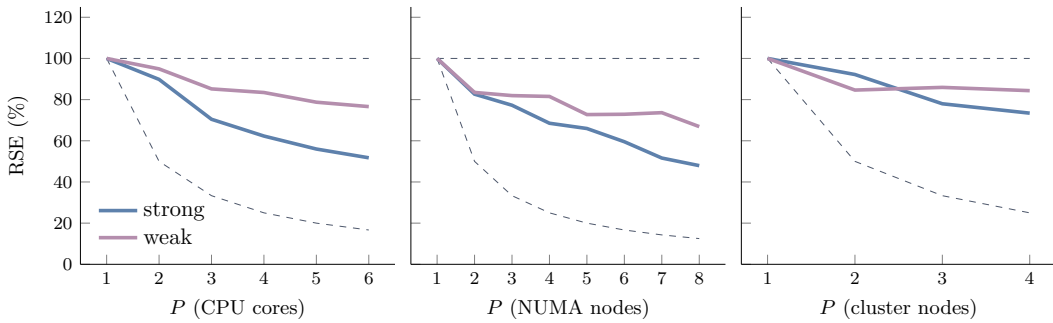
Both the strong and weak scalability of `accu_threshold` over the CPU cores within a NUMA node are higher than 80% (Table 5.5). When scaling over the NUMA nodes within a cluster node, the algorithm has more trouble of using additional workers effectively. Also, the pattern of efficiencies over the number of workers becomes irregular (Figure 5.6). The speed-up when using 8 NUMA nodes compared to using 1 is about 5.5.

To gain insights into why, in the case of scaling over NUMA nodes, the scaling efficiencies are irregular, we performed two additional experiments. The first experiment is targeted at determining to what extend the flow direction field used in the experiment is responsible for the irregularity. In the case of a weak scalability experiment, each time a worker is added, the data set used is increased in size. If flow accumulation calculations perform particularly well (or bad) on these newly added parts, then this will have an effect on the efficiencies for these workers. To determine the influence of the flow direction data on the pattern of scaling efficiencies, we performed the same strong and weak scalability experiments as presented in Figure 5.6a, but for a different part of the African continent (centered in the Sahara desert in Western Africa, instead of in Central Africa), and compared the corresponding scaling efficiencies. The resulting pattern of scaling efficiencies are comparable to each other (Figure 5.7), suggesting that differences in flow direction fields are not responsible for the irregular pattern.

The second experiment we performed is targeted at determining whether the procedure for assigning array partitions to processes is relevant for explaining the irregular pattern of scaling efficiencies. When a partitioned array is created, its partitions are instantiated in the different cooperating processes. Which one exactly depends on a



(a)



(b)

Figure 5.6: Relative scaling efficiencies (RSE) of experiments performed for `accu_threshold` (a) and the case-study model (b, Table 5.3). For reference, efficiencies for linear scaling (upper dashed line) and serial scaling (lower dashed line) are also shown.

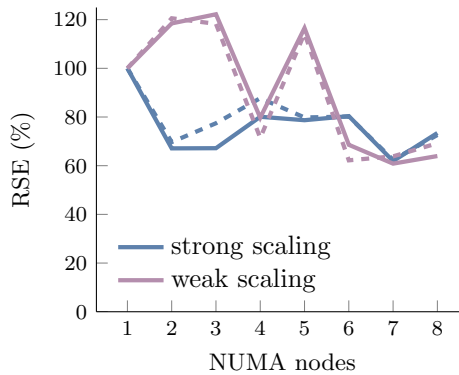


Figure 5.7: Relative strong and weak scaling efficiencies (RSE) of experiments performed for `accu_threshold`. The solid lines correspond with those from the original experiments shown in Figure 5.6a, and the dashed lines with results of the same experiments, but for a different flow direction field.

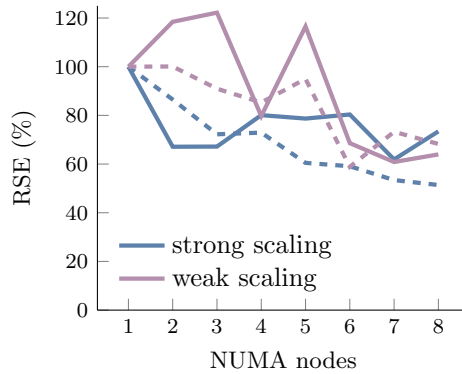


Figure 5.8: Relative strong and weak scaling efficiencies (RSE) of experiments performed for `accu_threshold`. The solid lines correspond with those from the original experiments shown in Figure 5.6a, and the dashed lines with results of the same experiments, but using a different procedure for assigning partitions to processes.

procedure for mapping 2D array partitions to a 1D array of process IDs. LUE uses the Hilbert curve for this, but can also use an alternative, like a linear mapping from 2D array partitions to 1D process IDs. An advantage of using the Hilbert curve is that partitions representing nearby areas in 2D space tend to be located in the same or a nearby process [65]. This reduces the latencies involved when tasks managing neighbouring partitions need to communicate with each other, like in the case of flow accumulation. Again, we performed the same strong and weak scalability experiments as presented in Figure 5.6a, but now using a linear (row-major) mapping from 2D array partitions to 1D process IDs instead of the Hilbert curve. The resulting pattern of scaling efficiencies are different from each other (Figure 5.8), suggesting that differences in the procedures for mapping array partitions to NUMA nodes is (partly) responsible for the irregular pattern.

To put the differences in scaling efficiencies into perspective we looked at the absolute performance when using all 8 NUMA nodes in a cluster node. When allocating partitions to processes using the Hilbert curve, the calculations performed by the strong scalability experiment were 27% faster (990 ms versus 1355 ms), whereas those performed by the weak scalability experiment were 16% slower (9455 ms versus 8153 ms).

In case of the case-study model, strong scaling efficiencies are lower than weak scaling efficiencies (Figure 5.6b). Given the latencies of the

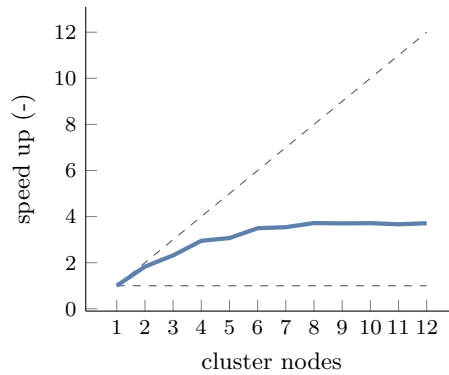


Figure 5.9: Speed-ups for the strong scaling experiment performed for the case-study model (Table 5.3). For reference, speed-ups for linear scaling (upper dashed line) and serial scaling (lower dashed line) are also shown.

model when using a whole cluster node, the scaling efficiencies are high—around 80%. Additional cluster nodes can be used effectively.

An additional experiment was performed to determine at how many cluster nodes scalability stops and what the maximum associated speed-up is. As described in Section 5.4.2, the number of workers over which scalability experiments can be performed is limited by the availability of a large flow direction data set without a lot of no-data cells. To provide information to modellers who are interested in the maximum speed-up that can be achieved, we performed the strong scalability experiment for the case-study model for an increasing number of cluster nodes. At 8 cluster nodes, the speed-up is almost 4 and does not increase anymore (Figure 5.9). When using that many nodes the amount of hardware is very large compared to the problem size. At 8 cluster nodes, the number of CPU cores used is 384, which is more than the number of array partitions per raster for which tasks are being created (121). To still be able to provide all CPU cores with enough work in such a situation, requires a lot of independent tasks. At some point there are not enough of those anymore, and the scalability stops.

5.5.3 Composability

Running the model from Listing 5.1 without synchronization takes less time than with synchronization (22s versus 25s). The difference is relatively small, but in simulation models in which many operations

are used these small performance gains may become relevant. Also, even a small load imbalance can cause workers to be drained of useful work to do, decreasing the scalability.

The traces in Figure 5.5 illustrate a result of a similar experiment, on a smaller dataset run on the 6 cores within a single NUMA node (Section 5.4.1). In case of no synchronization point between the operations, even though there is a data dependency between the two operations, tasks from the second call execute while those from the first operation are still executing as well. This is especially apparent for the tasks calculating NIDP values. Since these only depend on the flow direction raster passed in, tasks created by both calls to `accu_threshold` can be scheduled for execution immediately. In case of a synchronization point between the operations, this does not happen. Only once all tasks from the first call to `accu_threshold` are finished, can tasks from the second call be scheduled for execution. This results in a larger fraction of time that workers are drained of useful work to do, identified by the arrows in Figure 5.5, which results in longer model latencies.

5.6 DISCUSSION

Our new algorithms support the use of various criteria to determine how much of the total amount of material entering each cell remains in that cell as residue and how much flows towards the downstream cell. Compared to an existing implementation we compared the performance with, the single core performance of our algorithms is better. Given the scaling efficiencies of our new flow accumulation algorithms and the case-study model, we conclude that in case a modeller needs to decrease the latency of a model, or to use a model on a larger dataset, additional hardware can be used effectively. The use of AMT in the implementation of the algorithms supported the requirement that modelling operations should be composable. We showed that concurrent tasks from consecutive flow accumulation operations were scheduled to run in parallel. This led to an improvement of the overall latency, which is beneficial for the scalability of whole models.

5.6.1 *Performance and scalability*

We conclude that the single CPU core performance of the LUE flow accumulation algorithms is good, albeit details in the functionality of different implementations might be different. The `accu` and `accu_-`

threshold algorithms are more than twice as fast as the same operations in PCRaster.

The variation in the scaling efficiencies when scaling `accu_threshold` over NUMA nodes suggests that each time workers are added something changes in the way the total amount of work is performed. Additional experiments showed that this is not related to differences in the flow direction field used in the experiments, but is—at least partly—related to how array partitions are distributed over processes. Using a different procedure for this resulted in a different pattern of scaling efficiencies over NUMA nodes. When using all 8 NUMA nodes in a cluster node, in case of the strong scalability experiment, using the Hilbert curve clearly outperformed the linear mapping procedure, both in terms of the scaling efficiency and the absolute performance. In case of the larger problem solved by the weak scalability experiment, the relative scaling efficiency and the absolute performance are similar, but in favour of the linear mapping procedure.

We did not see a similar variability in scaling efficiencies when scaling `accu_threshold` over CPU cores. This is likely related to the differences in latencies between the NUMA nodes. Different combinations of NUMA nodes exchange data at different speeds, depending on their locations in the cluster nodes. In case of using the Hilbert curve procedure for mapping partitions to processes sometimes increases the performance and sometimes decreases the performance relative to what can be expected given the performance on a single NUMA node. Using the linear mapping procedure, the performance varies less per set of NUMA nodes.

In the scalability experiments of the case-study model, any irregular variability in performance over NUMA nodes is likely to be hidden by the much larger number of tasks that are performed. As long as there is enough useful work to do, the HPX runtime hides latencies involved in communication between processes.

5.6.2 *Composability*

The lack of unnecessary synchronization points within our flow accumulation algorithm results in a relatively large set of tasks that are ready to execute. This limits the negative effect of load imbalance on the overall latency, even when executing a single call to a flow accumulation operation. While a task managing a partition containing

a downstream part of a large scale stream is waiting for material to accumulate, there are likely other tasks that can do something useful.

5.6.3 *Future work*

Our results show that it is useful to apply the AMT approach to flow accumulation operations. Given this and our experiences with other operations [88], this suggests that it is useful to apply the approach to other modelling operations as well. This can potentially result in a set with which model developers can build a wide range of models that perform and scale well. To increase knowledge and experience when moving in this direction, in our view several aspects deserve attention. Transporting material using flow accumulation operations assumes that the duration of the simulated time step is longer than the material requires to reach the simulated area's outflow point. Other useful operations exist that do not require this and model flow routing in greater detail. Examples of these are the kinematic wave, diffusion wave and dynamic wave operations [161]. It is unclear how to express these operations using the AMT approach. Also, since these operations require more computations, the temporal load imbalance resulting from them will have a larger impact on the scalability than in the case of flow accumulation operations. To work around this, a procedure may be required to redistribute partitions over processes, based on load imbalance detected at runtime. Additionally, simulation models often require a lot of I/O to read and write model state to datasets. Traditional serial I/O prevents the scalability of models. Using a parallel file system and parallel I/O allows the I/O to be scalable, over I/O nodes. But it is unclear how to best integrate parallel I/O with the AMT approach.

Finally, there are at least two opportunities for further improving the performance of the existing modelling operations. First, as we showed, the procedure for assigning partitions to processes is of influence on the performance and scalability of operations. Which procedure works best may depend on characteristics of the hardware, like the actual differences in latencies between NUMA nodes. Additionally, it may depend on the input data. In the case of flow accumulation operations, grouping partitions depending on their membership of a hydrological catchment may be useful to improve its performance. A second opportunity to improve the performance of operations is to integrate the use of GPGPU devices often available to the model developer. It remains to be seen how to integrate them in a modelling

framework generating a different set of tasks for each model, and to what extent this increases the performance of models.

SYNTHESIS

As stated in the introduction, the main goal of this research is to gain knowledge required to be able to develop a modelling framework for building large models in which both discrete objects and continuous fields can be simulated. This thesis contributes to two aspects of such a framework. First, a single data model is used, to represent the state of the environment. Information about discrete objects and continuous fields is organized in a uniform way. Second, models created by the framework scale automatically, supporting many calculations on large datasets. Technical details related to how this works are hidden from model developers.

This chapter reflects on the research in the context of the main goal. Answers to the research questions mentioned in Section 1.4 are provided in Sections 6.1 and 6.2, as well as the implications the results have, and remaining challenges for which this research does not provide an answer yet.

6.1 REPRESENTING GEOGRAPHICAL STATE

6.1.1 *Answers and implications*

Research question 1A

How can the different data models currently used to represent spatio-temporal agents and fields be generalized into a single conceptual data model supporting the integrated simulation of both agents and fields?

In Chapter 2 a conceptual data model has been presented that is capable of representing model state of both agent- and field-based models. With this data model, seemingly different kinds of state variables can be regarded as being of the same kind, or simplifications thereof. All information related to a collection of objects is aggregated into a *phenomenon* (Figure 2.3). Within a phenomenon, information related to a collection of objects sharing a space and time *domain* is grouped into

property-sets. The domains contain information about when and where object-related information is defined. This can be object IDs, for example, to represent when an object was “born” or when it “died”. There are multiple approaches to represent locations in time, for example using time points or time intervals. Which one to use depends on the information that is referred to by the property-set. Similarly, there are multiple approaches to represent locations in space, for example using points, lines or polygons. The actual information within a property-set is located in *properties*. For each location in time, each property within a property-set contains a value for each object.

Several aspects of this data model make it different from existing data models for representing model state in agent- and field-based models. Some of these differences result from design decisions that were taken because of the use of the data model in the context of a modelling framework, and for performance reasons. To be able to build a modelling framework with pre-built modelling operations it is beneficial that there are as few dependencies as possible between the implementation of the operations and the data model. For example, an operation that only adds two different properties together can be implemented in terms of two arguments containing, for all objects, the property values to sum. There is no need for this operation to know the type of the objects to which the properties belong. That would decrease the genericity of the modelling operation. To allow generic modelling operations to be developed, that can be used in as many contexts as possible, it is beneficial that each kind of information related to a collection of objects is grouped together, in homogeneous collections. This is also beneficial with respect to the runtime performance of modelling operations. Due to the better spatial (data) locality [140], iterating over homogeneous collections of values results in fewer CPU cache misses than iterating over a collection of heterogeneous objects containing values (see also Section 3.3).

A first unique aspect of the data model, therefore, is that information related to objects, like their IDs, locations in space, and property values, is not grouped per object but per kind of information, in the various components of the conceptual data model. For example, all IDs of all objects are located at the level of the phenomenon (Figure 2.3), all locations in space of all objects are located in the space domain of a property-set, and all property values of all objects are located in a property of a property-set. This facilitates the use of state variables in a simulation model using high-level procedural mod-

elling operations. In an expression like `phenomenon.property_set.a + phenomenon.property_set.b`, two properties are involved (*a* and *b*), referring to all values of possibly large collection of objects. Generic modelling operations can be provided by a modelling framework that take such property values as arguments. Since such operations do not depend on agent types used in a specific agent-based model, they can be used by multiple models. Additionally, grouping the same object-related information allows many diverse kinds of information related to the same collection of objects to be aggregated. For example information that differs with respect to variability through time, variability through space, mobility, and lifespan can all be represented by a single state variable organized according to the conceptual data model. And, by standardizing the way object-related information is grouped and organized, there is no need anymore for model developers to define types for different kinds of agents in an agent-based model.

A second aspect of the conceptual data model that makes it different from other data models is the implicit assumption that information that is not explicitly linked with a time domain or space domain is considered omnipresent in time or space. For example, properties in a property-set without a space domain contain values that are valid for each location in space. Like properties in a property-set that do have a space domain these properties are spatial, but they are not bounded in space and they do not vary through space. The same principle can be applied to temporal information. This implicit assumption is what makes all state information spatio-temporal information.

The conceptual data model allows state variables from the agent-based and field-based modelling domains to be represented. All information related to a specific kind of agent, like trees or humans, can be represented by a single phenomenon. The locations in time and space are represented by the time and space domains in one or multiple property-sets, and the characteristics of the agents are represented by properties. Relations between agents are represented by a separate phenomenon containing “relation-objects”.

Information related to all fields in a field-based model can also be represented by a single phenomenon. This phenomenon contains a single object—representing the simulated spatial area—and each field is represented by a property in a property-set. Any scalar variables also representing a characteristic of the simulated area can be added to the same phenomenon. Organizing information according to the

conceptual data model removes the distinction traditionally made between representing agents and fields.

Besides being able to represent typical state variables from the agent- and field-based modelling domains, the conceptual data model can represent more kinds of state variables. For example, mobile spatial fields can be represented, by using a spatial domain that varies through time instead of a constant spatial domain. As another example, relations between objects can be represented that vary continuously through time and space. This allows relations to be simulated that are stronger at certain locations in time and space, and weaker in other locations. And the spatial discretization of property values can vary through time (and space) as well. This may be useful in case a process is simulated whose activity strongly varies through time (and space).

Representing all state variables (both agents and fields) in a simulation model using the same conceptual data model facilitates the design of an easy to use modelling framework. For example, when calculating the topographic slope of multiple areas, the developer can now call a (future) framework operation accepting a state variable representing the surface elevation of these areas using a statement like this: `earth.continent.slope = fr.slope(earth.continent.dem)` . This operation implicitly iterates over all areas to calculate the slope of each of them. As another example, a group of animals can be moved in a unique direction using a statement like this: `animals.location = fr.move(animals.location, offset)` , in which `offset` contains an offset for each animal. These example statements are similar to a typical field-based model expressed using map algebra, but a large amount of complex data may be referred to by the state variables.

Whenever appropriate, framework operations can be overloaded to support state variables referring to different kinds of information associated with a collection of objects. The model developer can then focus more on what the state variables mean in the domain of the simulated system, rather than having to consider the specific types. In principle this is not new. As described in Section 1.2.2, it is customary in field-based modelling using map algebra to combine rasters with scalars. With the conceptual data model, this same principle can be applied in the context of integrated simulation of both agents and fields.

Research question 1B

How can information related to large collections of spatio-temporal objects and fields be stored using a single physical data model?

In Chapter 3 a physical data model has been presented for storing state variables whose information is organized according to the conceptual data model presented in Chapter 2. One goal of the conceptual data model is to provide the model developer with a uniform way to organize each state variable. The underlying data referred to by state variables used in agent-based and field-based modelling varies greatly with respect to aspects as how the locations in time and space are represented, the variability of information through time and space, and whether or not objects are added and removed from the collection over time. These differences potentially result in an unfeasible large number of individual physical data models. The physical data model presented in this thesis is capable of representing the underlying data using three layers of abstraction.

With the first layer of abstraction, six kinds of arrays of temporal information related to objects can be stored. With the second one, layered on top of the first, basic information related to the conceptual data model can be stored: object identity, locations in time, locations in space, and properties. With the third layer of abstraction, layered on top of the second, collections of spatio-temporal objects can be stored. These represent the objects referred to by the state variables in a simulation model.

The three layers of abstraction are implemented in C++. The first abstraction layer is implemented in terms of the HDF5 data model [162] (Figure 3.1). The public API, for interacting with the physical data model, is implemented in terms of the third abstraction layer. This API is also accessible from Python.

Besides being able to represent discrete objects and continuous fields, the physical data model has several additional features that—when combined—set it apart from other data models used in spatio-temporal simulation modelling. Locations in simulated time can be represented in any resolution typically used in simulation modelling, ranging from nano-seconds to millions of years. Additionally, the physical data model does not require an associated server process and can be used on any platform commonly used for simulation modelling, including HPC facilities, and resulting datasets are portable between different

platforms. Finally, the amount of information stored in a data set is limited by properties of the hardware, not by an arbitrary limit. The LUE software implementing the physical data model is currently being used by the Campo modelling framework to store the simulated agents and fields [153].

The physical data model provides a unified approach to store information organized according to the conceptual data model. Given the approach of separate layers of abstraction, it is likely that information for which no support is available in the physical data model yet, can be added in the future.

6.1.2 *Remaining challenges*

Although the presented conceptual and physical data models support a uniform representation of different kinds of state variables, there are types of information that are used in simulation models which were not considered in detail. To be able to represent this information and to provide good performance, these need to be considered in future research. Important examples of these kinds of information are 1) relations between objects, within the same phenomenon and between different phenomena, 2) uncertainty in locations in time and space, and in property values, 3) additional kinds of space domain items, like lines and polygons, and their topological relations, and 4) a spatial index to be able to filter information by a bounding box, for example, without having to read all data. For all of these, a mapping must be created between the raw information and the representation of it in terms of arrays of temporal information (abstraction level 2 of the physical data model, see Section 3.3.2).

The prototype modelling language presented in Chapter 2 and developed further as part of the Campo modelling framework [153], represents a first step towards a model development interface for simulating systems of agents and fields. Several aspects of such a language remain to be considered. For example, given a data model that is able to uniformly represent state variables that actually refer to very diverse kinds of data, and a set of modelling operations, it may not always be clear what the operations do. For example, assuming `park.area.biomass` refers to the biomass raster property of a collection of parks, and `deer.location.weight` refers to the weight scalar property of a collection of deer, it is not immediately clear what the addition operation in the expression `park.area.biomass + deer.-`

`location.weight` calculates and returns. There are two spatial domains involved, of the `park.area` and `deer.location` property-sets. The operation could return new rasters, new points, or fail because of the ambiguity. Perhaps certain operations should only be allowed when information referred to by the arguments are part of the same phenomenon, or even the same property-set. A set of simple rules is needed that makes it clear what to expect when passing state variables to operations. Further research is needed to define these.

With respect to the physical data model, additional research is needed to make performing I/O scale with additional I/O nodes. The HDF5 data model used by the physical data model provides a parallel I/O API for performing I/O to a parallel file system. Given the many kinds of information that can be represented by the physical data model, it is yet unclear what a good strategy is to achieve scalable I/O in all cases, and how to hide the complexity related to performing parallel I/O from the user of the data model.

6.2 SCALABLE COMPUTING

6.2.1 *Answers and implications*

Research question 2A

How can the AMT approach be applied in the implementation of a modelling framework which results in models that perform well and scale well over CPU cores and cluster nodes?

In the research presented in Chapters 4 and 5 the AMT approach is applied in the design and implementation of a field-based modelling framework containing a set of local, focal, zonal, global, and flow routing operations. When developing a modelling framework for scalable models, there are multiple aspects the framework developer must consider. Since the model developer using the framework will combine individual building blocks to define models, the framework developer must design these building blocks in such a way that work is automatically distributed over the available workers, and that the resulting models perform and scale well. The AMT approach as supported by the HPX library lends itself well for this because it hides much of the complexity usually involved with implementing parallel and distributed software. All hardware, parallel and distributed or not, is presented as a single abstract machine, consisting of a num-

ber of cooperating processes. Each of these processes contains task queues and a task scheduler per CPU core. As long as there are enough tasks ready to run, all CPU cores will be busy doing useful work. The responsibility of the framework developer is to define the tasks and the dependencies between them, and distribute these evenly over the processes.

Parallelization and distribution of modelling operations working on large datasets requires the data to be distributed as well. In the approach presented in this thesis, the spatial domain is decomposed into rectangular partitions. Expressing modelling operations in terms of partitioned arrays results in the flexibility required when developing for an unknown number of cooperating processes with an unknown number of CPU cores, and an unknown amount of memory. Since the partition size can be configured, it is possible to optimize task sizes and the number of tasks to achieve the best performance given a specific hardware configuration. If the partition size were dependent on the data, like the shape of hydrological catchments for example, this would not be possible.

Implementing the partitioned array as a collection of futures to array partitions makes the type useful for representing rasters that are translated by modelling operations. In general, the responsibility of a field-based modelling operation then is to attach asynchronous continuations to the futures to array partitions, and return a new partitioned array containing futures to new array partitions. Because of the asynchrony, defining the model and actually executing the model become separated from each other. As described earlier, this has the advantage of being able to provide the hardware with more useful work to do. Since modelling operations only need to *define* the work to be performed, rather than *executing* the work, executing the modelling operations of a model finishes much faster than executing the work defined by those operations.

An aspect of AMT that is especially useful in a modelling framework is the fact that it is possible for tasks from subsequent modelling operations to execute in parallel. This is a consequence of the fact that tasks are spawned asynchronously, which results in a future object to each task's result, and that continuations can be attached to such a future object. This approach makes it convenient to supply a computer or a computer cluster with tasks from multiple modelling operations. Whatever the combination of modelling operations, if there is poten-

tial for concurrent tasks created by different operations to execute in parallel, then this may happen.

Our experimental results show that operations and case-study models built with a modelling framework using AMT work well. Performance and scalability are both good in general. The framework's modelling operations can be developed independently, as free functions, and combinations of these operations work well together. There is therefore no need for an interpreter to analyse and optimize models. This simplifies the framework and allows the operations to be called from existing high-level scripting languages, like Python.

Care must be taken though, to the total number of continuations attached by modelling operations. In simulations involving many operations on large amounts of data, the data structures used by HPX to keep track of futures and their continuations can become large. In such cases it may be necessary to limit the amount of memory required by these data structures. A solution for this is to explicitly keep track of how many operations have attached continuations to futures, and temporarily pause calling modelling operations when some threshold is reached.

In Chapter 4 the usability of the AMT approach to the framework developer was evaluated. The biggest difference compared with alternatives like using MPI and OpenMP, is that with a modelling framework implemented using AMT, most data is referred to by futures. All calculations must be expressed as continuations to (compositions of) futures. The ParalleX execution model is very different from the ones implemented by MPI and OpenMP, for example. The framework developer has to become familiar with it, of course, and be up to date with the relatively recent additions to the C++ standard. The learning curve associated with using AMT/HPX can be steep, depending on the experience of the framework developer.

Research question 2B

How can the AMT approach be applied in the implementation of a set of parallel and distributed flow accumulation operations that perform well and scale well over CPU cores and cluster nodes?

In Chapter 5 details of a set of flow accumulation algorithms implemented using AMT have been presented. Each of these algorithms uses a similar scheme, in which a single task per input array partition

is defined, which has the responsibility to do whatever it takes to calculate the flow accumulation result for that partition. Each of these tasks uses additional tasks for performing different kinds of work. The result is a calculation that is performed asynchronously and without any tasks having to wait for information that is not of its concern. Tasks are only dependent on tasks managing directly neighbouring partitions.

There are a number of benefits of this approach. For a large part, load imbalance within each process is hidden by tasks of the flow accumulation operation itself. Tasks managing partitions containing downstream parts of a stream network are skipped until the information that is required for them to execute becomes available. In the meantime, tasks managing partitions containing upstream parts of a stream network are scheduled for execution by the AMT runtime. Because modelling operations are composable, any remaining load imbalance can be hidden by tasks from subsequent operations in the model.

Another benefit of the approach is that for the framework developer, there is less complexity to handle. There is no need to consider solving the flow accumulation problem for the whole spatial domain. The problem is limited to solving the problem for a single partition. This may well be an approach that can be used for many other modelling operations as well. Because of the “loose connections” between tasks managing neighbouring partitions and the absence of connections between tasks managing non-neighbouring partitions, the AMT runtime has a good opportunity to automatically hide any load imbalances, at least within each process.

The results of the experiments presented in Chapter 5 show that the performance and scalability of the new flow accumulation operations is good. It is useful to increase the amount of hardware used to execute the operations if a result must be calculated in less time or if a result must be calculated for a larger raster.

6.2.2 *Remaining challenges*

As mentioned in Chapter 5, the AMT runtime of the HPX library supports stealing tasks from queues within a each process. This is convenient in case of load imbalance within processes. In reality however, there will also be load imbalance between processes, for example due to a non-uniform spatial distribution of no-data values. This can be solved by the modelling framework, by monitoring the load imbalance

between processes and migrating array partitions from busy processes to less busy processes. There is a cost involved in this migration, so additional research is required to define a heuristic that results in a good adaptive response to such load imbalance.

The modelling operations we considered have a relatively low computational intensity: once array elements arrive at the CPU core's register, only one or a few computations are performed using these elements as inputs, before moving to other elements. Given the difference between the latencies involved in accessing array elements from main memory and the upper CPU cache levels, and the speed with which a CPU core can do computations, the throughput of these operations is memory I/O bound. In general, this limits their performance, and within a NUMA node their scalability as well, compared to operations that have a higher computational intensity. This problem is hard to solve for operations that happen to have a low computational intensity, but there is an opportunity for replacing sets of much-used combinations of operations by single operations. For example, models calculating the normalized difference vegetation index (NDVI) based on two bands from a remote sensing image, always perform the same calculations: $(\text{near_infrared} - \text{red}) / (\text{near_infrared} + \text{red})$. For such a case an `ndvi` operation can be implemented, increasing the computational intensity (although not much), and hence the performance and possibly the scalability. Another example is the Penman equation for calculating evaporation [161], which is used in many hydrological models. It is likely that various modelling domains have their own set of frequently used combinations of basic operations. Integrating frequently used combinations of operations into their own operations also renders the need for—potentially large—temporary data structures that would otherwise be needed. An `ndvi` operation does not need one, whereas without such an operation two are needed (one for the numerator and one for the denominator).

Replacing combinations of basic operations by more complex ones arguably goes against the idea of a modelling framework consisting of generic building blocks. The frequency with which a certain combination of operations is used, and the impact of any performance and scalability issues, can be used as a guideline to determine whether it is worth the effort to implement an additional operation.

Given the good results of using AMT in the context of a field-based modelling framework, a future challenge is to apply this approach to modelling operations supporting the creation of agent-based models.

This would probably require a data structure for evenly distributing information related to collections of agents. One way to achieve this is to use the spatial distribution of the agents to guide the distribution of this information, much like how this is done with the partitions of the partitioned array presented in Chapter 4. This has the advantage that when agents need to sense each other or their environment, this information is likely nearby already, in the same process. But this approach assumes that agents are likely to be evenly distributed over space and that this distribution stays that way during the simulation. It is unlikely that this is often the case. Distributing information about spatially clustered agents evenly over the processes will result in increased network latencies, but not distributing it evenly results in more load imbalance. Further research is needed to find a good approach for distributed computing with large sets of mobile agents, in the context of information about the environment that is spatially continuous and stationary.

To make the current LUE framework useful for general use in field-based simulation modelling the existing set of modelling operations can be extended with more operations. Various local, focal, zonal, and global operations are still missing, but can be added without much effort. Operations working on a cost friction-distance path surface [24] and flow routing operations that take flow velocity into account [161] are more involved, but necessary to end up with a complete set of map algebra operations. As a first step, to make the framework useful in the domains of global hydrological modelling and global land-use change modelling, additional AMT-based flow routing algorithms are currently being designed.

6.3 INTEGRATED AGENT- AND FIELD-BASED MODELLING

The previous sections have focused on answers on the research questions that the research described in this thesis tries to answer, their implications, and remaining research topics that are related to that research. In this section we broaden the scope to the main goal to which this research contributes, namely to gain knowledge required to be able to develop a modelling framework for simulating large spatio-temporal systems of agents and fields.

6.3.1 *Implications*

The research presented in this thesis is especially relevant for framework developers. It provides information that can be used for the design and development of a modelling framework for simulating large spatio-temporal systems of agents and fields. The conceptual and physical data models are defined in such a way, that they can be extended to include additional information, possibly for specific modelling domains and applications.

Given the data models and the fact that fields are represented by it as properties of objects, any field-based model can be considered to be an agent-based model. In a field-based model the state of a single anonymous object (the simulated area) is simulated, and this state is represented by rasters and scalars. This insight provides a path towards a modelling framework that generalizes both agent- and field-based modelling frameworks. Agents and fields may not be that different as they are generally considered to be. For practical purposes, and within the domain of spatio-temporal simulation modelling, fields can be considered to be a subset of agents.

AMT has proven to be a good approach for use in a modelling framework. Good performance and scalability can be achieved. By hiding much of the underlying technology related to assigning work to parallel and distributed hardware, the framework developer is freed of much of the complexity associated with that, and can focus on the modelling operations themselves.

A major benefit of using AMT for the model developer is that, because of the asynchrony, the execution model of the model is separated from the execution model of the modelling operations. Although the model definition is executed according to the execution model of the host language, like Python or C++, the tasks generated by the modelling operations are executed according to the execution model of the AMT runtime. The model developer can use a familiar programming language to define seemingly serial models, while the framework developer has the freedom to use the potential of the AMT facilities to express the work to be performed, using all hardware available at runtime.

Given the above, there is no need for the modeller to run large models on multiple subsets of the spatial domain. The effort required to perform large modelling studies can become similar to executing a small one. Additionally, because with the presented data models all

information related to the same objects is aggregated, and stored in a single dataset, the complexity with respect to data management is reduced. All information about certain kinds of objects is aggregated within a single dataset, and all model state can—optionally—be stored in a single dataset.

6.3.2 *Remaining challenges*

This research does not provide all knowledge required to develop a framework for simulating large spatio-temporal systems of agents and fields. Several aspects of such a framework remain to be investigated, either related to modelling concepts or to the framework design.

A framework supporting the development of integrated agent- and field-based models must provide the model developer with an easy to use uniform model development interface. In Chapter 1 it was mentioned that agent-based models are typically expressed using the object-oriented (OO) programming paradigm, while field-based models are typically expressed using the procedural programming (PP) paradigm. In Listing 6.1 and Listing 6.2 simplified examples of two user-defined operations are shown, implemented using those two paradigms. When OO is used, a modelling operation is a member function of an agent class. Assuming global variables are not used, the model developer can implement the operation in terms of variables passed in as arguments, and member variables. In case of using PP, the model developer must implement the operation in terms of variables passed in as arguments. Because of these differences, to the model developer free functions are easier to write and reason about. They are easier to write because OO requires more syntax to be able to refer to member variables. For example, in Python member variables need to be qualified by `self.`, while in C++ they sometimes need to be qualified by `this->`. Besides having to know about the syntax for defining classes, constructors and member functions, this is an additional burden on the model developer. Pure free modelling functions are easier to reason about because only local information is used—passed in as arguments—and they have no state or side effects. Since maintenance of software is expensive, it matters whether software is perceived as complex or simple.

Listing 6.1: Perform a user-defined operation on a collection of agents using the object-oriented programming paradigm.

```
class Agent(fr.Agent):
    def __init__(self):
        # ...

    def my_operation(self):
        self.property_a = \
            fr.an_operation(self.property_a, self.property_b)

for i in range(nr_time_steps):
    for agent in agents:
        agent.my_operation()
```

Listing 6.2: Perform a user-defined operation on a collection of agents using the procedural programming paradigm.

```
def my_operation(property_a, property_b):
    return fr.an_operation(property_a, property_b)

for i in range(nr_time_steps):
    agents.property_a = my_operation(
        agents.property_a, agents.property_b)
```

A result of using OO for expressing models for the framework developer is that it may limit the opportunity for frameworks to provide support. As shown in Listing 6.1, `fr.an_operation` is passed in two variables (`self.property_a`, and `self.property_b`), which are the individual member variables of a single agent. The same operation in Listing 6.2 is also passed in two variables, but each of these refers to all values for all agents. In the latter case, since the operation has to perform more work, the implementer of `an_operation` has more opportunity to optimize the operation, for example by parallelization.

Note that we are only contrasting OO versus PP as a means to implement simulation models. Given this limited scope, there is a case to be made for a model development interface that uses PP instead of OO, also for expressing agent-based models. In general, it is beneficial for both the user and the developer of a modelling framework when the responsibility of the framework is maximized. In our view, the model developer should not have to explicitly know about and use the exact types of the state variables, but only pass them into modelling operations which return new information. When the framework is updated, for example to support scalable models, models created with the framework should not have to be changed. Similarly, when the

state variables are updated, for example, to account for temporal or spatial variability, models created with the framework should also not have to be changed. Whether or not this ideal can be achieved is a topic for future research.

Part III

APPENDIX

The research presented in this thesis has resulted in version 0.3.0 of the LUE software [89], consisting of two parts. The first is called LUE data model, and contains all code related to the physical data model presented in Chapter 3. The C++ and Python APIs support performing I/O of model state variables to HDF5 data sets. The LUE data model is currently already being used in the Campo environmental modelling framework [153]. The second part is called LUE framework, and contains all code related to the modelling framework and operations presented in Chapters 4 and 5. The C++ and Python APIs allow for scalable models to be created. Work is currently ongoing to make the LUE framework useful in the context of global hydrological modelling. This involves completing an initial set of modelling operations.

A feature of the modelling algorithms that has not been focused on in this thesis is that they are configurable at compile-time. A policy-based design is used to make the algorithms useful in multiple contexts, and to make them efficient. Elementary aspects that are configurable are 1) how to detect no-data values in the arguments of algorithms, 2) how to detect values that fall outside of the algorithm's valid domain, 3) how to detect values that fall outside of the output type's representable range, and 4) how to mark no-data values in the result. Some algorithms can have additional policies, for example in the case of focal operations for handling halo cells, that fall outside of the extent of a raster.

The idea is that some functionality of the modelling algorithms is optional, and in case it is not required, the compiled algorithm will not contain that functionality and does not lose performance because of it. Additionally, in the different contexts in which the algorithms are used, different conventions may be used, for example with respect to the encoding of no-data. The LUE modelling algorithms can be compiled with support for such alternative conventions.

LUE is free to use by model developers. It can be installed on Linux, MacOS, and Windows. The source code is licensed according to an open source license. The development team welcomes researchers and software developers who want to cooperate to the development of the

software. More information, including documentation and a support channel can be found on the LUE website (Table A.1).

home page	lue.computationalgeography.org
source code	github.com/computationalgeography/lue
version 0.3.0	K. de Jong and O. Schmitz. <i>computationalgeography/lue: LUE-0.3.0: Scientific database and environmental modelling framework</i> . Version 0.3.0. Sept. 2021. doi: 10.5281/zenodo.5535686

Table A.1: More information about the LUE software.

The research presented in the various chapters of this thesis is based upon intermediate versions of the software. Information about these versions can be found in Table A.2.

Chapter	More information
2	pcraster.geo.uu.nl/misc/developments/generic-modelling-of-fields-and-agents , github.com/pcraster/datamodel_prototype
3	github.com/pcraster/paper_2019_physical_data_model
4	github.com/computationalgeography/paper_2020_scalable_algorithms
5	github.com/computationalgeography/paper_2021_routing

Table A.2: URLs to more information about the software presented in the various chapters of this thesis.

BIBLIOGRAPHY

- [1] M. Abdou, L. Hamill, and N. Gilbert. "Designing and building an agent-based model." In: *Agent-Based Models of Geographical Systems*. 2012, pp. 141–165. DOI: 10.1007/978-90-481-8927-4_8.
- [2] R. Alsaleh and T. Sayed. "Modeling pedestrian-cyclist interactions in shared space using inverse reinforcement learning." In: *Transportation Research Part F: Traffic Psychology and Behaviour* 70 (2020), pp. 37–57. DOI: 10.1016/j.trf.2020.02.007.
- [3] Altrema. *Adaptive Modeler*. URL: <https://www.altrema.com> (visited on 01/01/2016).
- [4] L. An, V. Grimm, A. Sullivan, B. T. II, N. Malleson, A. Heppenstall, C. Vincenot, D. Robinson, X. Ye, J. Liu, E. Lindkvist, and W. Tang. "Challenges, tasks, and opportunities in modeling agent-based complex systems." In: *Ecological Modelling* 457 (2021), p. 109685. DOI: 10.1016/j.ecolmodel.2021.109685.
- [5] L. An, J. Mak, S. Yang, R. Lewison, D. A. Stow, H. L. Chen, W. Xu, L. Shi, and Y. H. Tsai. "Cascading impacts of payments for ecosystem services in complex human-environment systems." In: *Journal of Artificial Societies and Social Simulation* 23.1 (2020). DOI: 10.18564/jasss.4196.
- [6] I. N. Athanasiadis and F. Villa. "A roadmap to domain specific programming languages for environmental modeling: key requirements and concepts." In: *Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling*. 2013, pp. 27–32. DOI: 10.1145/2541928.2541934.
- [7] D. Ayllón, S. F. Railsback, S. Vincenzi, J. Groeneveld, A. Almodóvar, and V. Grimm. "InSTREAM-Gen: Modelling eco-evolutionary dynamics of trout populations under anthropogenic environmental change." In: *Ecological Modelling* 326 (2016), pp. 36–53. DOI: 10.1016/j.ecolmodel.2015.07.026.
- [8] M. P. de Bakker, K. de Jong, O. Schmitz, and D. Karssenbergh. "Design and demonstration of a data model to integrate agent-based and field-based modelling." In: *Environmental Modelling*

- & Software* 89 (2017), pp. 172–189. DOI: 10.1016/j.envsoft.2016.11.016.
- [9] R. Barnes. “Parallel non-divergent flow accumulation for trillion cell digital elevation models on desktops or clusters.” In: *Environmental Modelling & Software* 92 (2017), pp. 202–212. DOI: 10.1016/j.envsoft.2017.02.022.
 - [10] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. “The multidimensional database system RasDaMan.” In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. 1998, pp. 575–577. DOI: 10.1145/276304.276386.
 - [11] P. Baumann. “Management of multidimensional discrete data.” In: *The VLDB Journal* 3.4 (1994), pp. 401–444. DOI: 10.1007/bf01231603.
 - [12] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. “Spatio-temporal retrieval with RasDaMan.” In: *Proceedings of the 25th VLDB Conference*. 1999, pp. 746–749.
 - [13] J. Becla, D. Zhang, M. Stonebraker, and P. Brown. “SciDB: A database management system for applications with complex analytics.” In: *Computing in Science & Engineering* 15 (2013), pp. 54–62. DOI: 10.1109/mcse.2013.19.
 - [14] D. A. Bennett and W. Tang. “Modelling adaptive, spatially aware, and mobile agents: elk migration in Yellowstone.” In: *International Journal of Geographical Information Science* 20.9 (2006), pp. 1039–1066. DOI: 10.1080/136588106000830806.
 - [15] T. Berger. “Agent-based spatial models applied to agriculture: a simulation tool for technology diffusion, resource use changes and policy analysis.” In: *Agricultural Economics* 25.2-3 (2001), pp. 245–260. DOI: 10.1111/j.1574-0862.2001.tb00205.x.
 - [16] R. A. Betts, L. Alfieri, C. Bradshaw, J. Caesar, L. Feyen, P. Friedlingstein, L. Gohar, A. Koutroulis, K. Lewis, C. Morfopoulos, L. Papadimitriou, K. J. Richardson, I. Tsanis, and K. Wyser. “Changes in climate extremes, fresh water availability and vulnerability to food insecurity projected at 1.5°C and 2°C global warming with a higher-resolution global climate model.” In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 376 (2018). DOI: 10.1098/rsta.2016.0452.

- [17] M. F. P. Bierkens, V. A. Bell, P. Burek, N. Chaney, L. E. Condon, C. H. David, A. de Roo, P. Döll, N. Drost, J. S. Famiglietti, M. Flörke, D. J. Gochis, P. Houser, R. Hut, J. Keune, S. Kollet, R. M. Maxwell, J. T. Reager, L. Samaniego, E. Sudicky, E. H. Sutanudjaja, N. van de Giesen, H. Winsemius, and E. F. Wood. "Hyper-resolution global hydrological modelling: what is next?" In: *Hydrological Processes* 29.2 (2014), pp. 310–320. doi: 10.1002/hyp.10391.
- [18] M. Bithell, J. Brasington, and K. Richards. "Discrete-element, individual-based and agent-based models: tools for interdisciplinary enquiry in geography?" In: *Geoforum* 39.2 (2008), pp. 625–642. doi: 10.1016/j.geoforum.2006.10.014.
- [19] R. S. Bivand, E. Pebesma, and V. Gómez-Rubio. *Applied Spatial Data Analysis with R*. Springer New York, 2013. doi: 10.1007/978-1-4614-7618-4.
- [20] E. Bonabeau. "Agent-based modeling: methods and techniques for simulating human systems." In: *Proceedings of the National Academy of Sciences* 99.suppl. 3 (2002), pp. 7280–7287. doi: 10.1073/pnas.082080899.
- [21] D. G. Brown, R. Riolo, D. T. Robinson, M. North, and W. Rand. "Spatial process and data models: toward integration of agent-based models and GIS." In: *Journal of Geographical Systems* 7.1 (2005), pp. 25–47. doi: 10.1007/s10109-005-0148-5.
- [22] P. G. Brown. "Overview of sciDB: large scale array storage, processing and analysis." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. 2010, pp. 963–968. doi: 10.1145/1807167.1807271.
- [23] P. Burek, J. van der Knijff, and A. de Roo. *LISFLOOD – Distributed Water Balance and Flood Simulation Model – Revised User Manual*. JRC78917, EUR 26162. Publications Office of the European Union, 2013. doi: 10.2788/24982.
- [24] P. Burrough, R. A. McDonnell, and C. D. Lloyd. *Principles of Geographical Information Systems*. 3rd ed. Oxford University Press, 2015.
- [25] G. Câmara, A. M. V. Monteiro, J. A. Paiva, J. Gomes, and L. Velho. "Towards a unified framework for spatial data models." In: *Journal of the Brazilian Computer Society* 7.1 (2000), pp. 17–25. doi: 10.1590/s0104-65002000000200003.

- [26] G. Câmara, D. Palomo, R. Cartaxo, and M. D. Souza. "Towards a generalized map algebra: principles and data types." In: *Brazilian Symposium on GeoInformatics*. 2005, pp. 66–81.
- [27] J. Carabaño, J. Westerholm, and T. Sarjakoski. "A compiler approach to map algebra: automatic parallelization, locality optimization, and GPU acceleration of raster spatial analysis." In: *GeoInformatica* 22.2 (2017), pp. 211–235. doi: 10.1007/s10707-017-0312-3.
- [28] J. C. Castilla-Rho, G. Mariethoz, R. Rojas, M. S. Andersen, and B. F. J. Kelly. "An agent-based platform for simulating complex human-aquifer interactions in managed groundwater systems." In: *Environmental Modelling & Software* 73 (2015), pp. 305–323. doi: 10.1016/j.envsoft.2015.08.018.
- [29] C. J. Castle and A. T. Crooks. *Principles and concepts of agent-based modelling for developing geospatial simulations*. Tech. rep. UCL Centre for Advanced Spatial Analysis, 2006.
- [30] B. Chamberlain, D. Callahan, and H. Zima. "Parallel programmability and the Chapel language." In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. doi: 10.1177/1094342007078442.
- [31] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. "X10: an object-oriented approach to non-uniform cluster computing." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2005, pp. 519–538. doi: 10.1145/1094811.1094852.
- [32] K. Clarke, J. Brass, and P. Riggan. "A cellular automaton model of wildfire propagation and extinction." In: *Photogrammetric Engineering & Remote Sensing* 60.11 (1994), pp. 1355–1367.
- [33] K. C. Clarke. "Mapping and modelling land use change: an application of the SLEUTH model." In: *Landscape Analysis and Visualisation: Spatial Models for Natural Resource Management and Planning*. Ed. by C. Pettit, W. Cartwright, I. Bishop, K. Lowell, D. Pullar, and D. Duncan. 2008, pp. 353–366. doi: 10.1007/978-3-540-69168-6_17.
- [34] N. Collier and M. North. "Parallel agent-based simulation with Repast for high performance computing." In: *SIMULATION* 89.10 (2012), pp. 1215–1235. doi: 10.1177/0037549712462620.

- [35] G. Cordonnier, B. Bovy, and J. Braun. "A versatile, linear complexity algorithm for flow routing in topographies with depressions." In: *Earth Surface Dynamics* 7.2 (2019), pp. 549–562. doi: 10.5194/esurf-7-549-2019.
- [36] T. J. Cova and M. F. Goodchild. "Extending geographical representation to include fields of spatial objects." In: *International Journal of Geographical Information Science* 16.6 (2002), pp. 509–532. doi: 10.1080/13658810210137040.
- [37] A. T. Crooks and C. J. E. Castle. "The integration of agent-based modelling and geographical information for geospatial simulation." In: *Agent-Based Models of Geographical Systems*. Ed. by A. J. Heppenstall, A. T. Crooks, L. M. See, and M. Batty. 2012, pp. 219–251. doi: 10.1007/978-90-481-8927-4_12.
- [38] A. T. Crooks and A. B. Hailegiorgis. "An agent-based modeling approach applied to the spread of cholera." In: *Environmental Modelling & Software* 62 (2014), pp. 164–177. doi: 10.1016/j.envsoft.2014.08.027.
- [39] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming." In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.
- [40] K. de Jong, D. Panja, D. Karssenberg, and M. van Kreveld. "Scalability and composability of flow accumulation algorithms based on asynchronous many-tasks." In: *Computers & Geosciences* 162 (2022), p. 105083. doi: 10.1016/j.cageo.2022.105083.
- [41] P. Degenne, D. Lo Seen, D. Parigot, R. Forax, A. Tran, A. Ait Lahcen, O. Curé, and R. Jeansoulin. "Design of a domain specific language for modelling processes in landscapes." In: *Ecological Modelling* 220.24 (2009), pp. 3527–3535. doi: 10.1016/j.ecolmodel.2009.06.018.
- [42] A. van Deursen, P. Klint, and J. Visser. "Domain-specific languages: an annotated bibliography." In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36. doi: 10.1145/352029.352035.
- [43] W. van Deursen. "Geographical Information Systems and Dynamic Models: Development and Application of a Prototype Spatial Modelling Language." PhD thesis. the Netherlands: Utrecht University, 1995. ISBN: 90-6809-206-5.

- [44] W. van Deursen, C. Wesseling, D. Karssenberg, K. de Jong, and O. Schmitz. *The PCRaster environmental modelling framework*. Version 4.2.1. URL: <https://pcraster.computationalgeography.org> (visited on 12/15/2021).
- [45] R. A. van Engelen. "ATMOL: A domain-specific language for atmospheric modeling." In: *Journal of Computing and Information Technology* 9.4 (2001), pp. 289–303. doi: 10.2498/cit.2001.04.02.
- [46] Esri. *ArcGIS Desktop*. URL: <https://www.esri.com> (visited on 12/12/2021).
- [47] R. Fabian. *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules*. Richard Fabian, 2018.
- [48] K. R. Ferreira, G. Camara, and A. M. V. Monteiro. "An algebra for spatiotemporal data: from observations to events." In: *Transactions in GIS* 18.2 (2014), pp. 253–269. doi: 10.1111/tgis.12030.
- [49] T. Filatova, P. Verburg, D. Parker, and C. Stannard. "Spatial agent-based models for socio-ecological systems: challenges and prospects." In: *Environmental Modelling & Software* 45 (2013), pp. 1–7. doi: 10.1016/j.envsoft.2013.03.017.
- [50] P. F. Fisher. "Models of uncertainty in spatial data." In: *Geographical Information Systems*. Ed. by P. Longley, M. Goodchild, D. Maguire, and D. Rhind. 1999, pp. 191–205.
- [51] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.
- [52] A. U. Frank. "Map algebra extended with functors for temporal data." In: *Perspectives in Conceptual Modeling*. 2005, pp. 194–207. doi: 10.1007/11568346_22.
- [53] J. G. Freire and C. C. DaCamara. "Using cellular automata to simulate wildfire propagation and to assist in fire management." In: *Natural Hazards and Earth System Sciences* 19.1 (2019), pp. 169–179. doi: 10.5194/nhess-19-169-2019.
- [54] A. Galton. "A formal theory of objects and fields." In: *Spatial Information Theory*. 2001, pp. 458–473. doi: 10.1007/3-540-45424-1_31.

- [55] A. Galton. "Fields and objects in space, time, and space-time." In: *Spatial Cognition & Computation* 4.1 (2004), pp. 39–68. doi: 10.1207/s15427633scc0401_4.
- [56] A. Galton and M. Worboys. "Processes and events in dynamic geo-networks." In: *GeoSpatial Semantics*. Ed. by M. Rodriguez, I. Cruz, S. Levashkin, and M. Egenhofer. 2005, pp. 45–59. doi: 10.1007/11586180_4.
- [57] T. Garcia de Senna Carneiro, P. Ribeiro de Andrade, G. Câmara, A. Miguel Vieira Monteiro, and R. Reis Pereira. "An extensible toolbox for modeling nature-society interactions." In: *Environmental Modelling & Software* 46 (2013), pp. 104–117. doi: 10.1016/j.envsoft.2013.03.002.
- [58] GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation. URL: <https://gdal.org> (visited on 12/15/2021).
- [59] S. Gebbert and E. Pebesma. "A temporal GIS for field based environmental modeling." In: *Environmental Modelling & Software* 53 (2014), pp. 1–12. doi: 10.1016/j.envsoft.2013.11.001.
- [60] M. F. Goodchild. "Prospects for a space-time GIS: space-time integration in geography and GIScience." In: *Annals of the Association of American Geographers* 103.5 (2013), pp. 1072–1077. doi: 10.1080/00045608.2013.792175.
- [61] M. F. Goodchild. "Field-based spatial modeling." In: *Encyclopedia of Database Systems*. 2009, pp. 1132–1138. doi: 10.1007/978-0-387-39940-9_163.
- [62] M. F. Goodchild, M. Yuan, and T. J. Cova. "Towards a general theory of geographic representation in GIS." In: *International Journal of Geographical Information Science* 21.3 (2007), pp. 239–260. doi: 10.1080/13658810600965271.
- [63] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2002.
- [64] N. Gorelick, M. Hancher, M. Dixon, S. Ilyushchenko, D. Thau, and R. Moore. "Google Earth Engine: planetary-scale geospatial analysis for everyone." In: *Remote Sensing of Environment* 202 (2017), pp. 18–27. doi: 10.1016/j.rse.2017.06.031.
- [65] C. Gotsman and M. Lindenbaum. "On the metric properties of discrete space-filling curves." In: *IEEE Transactions on Image Processing* 5.5 (1996), pp. 794–797. doi: 10.1109/83.499920.

- [66] J. B. Gregersen, P. J. A. Gijssbers, and S. J. P. Westen. "OpenMI: Open Modelling Interface." In: *Journal of Hydroinformatics* 9.3 (2007), pp. 175–191. doi: 10.2166/hydro.2007.023.
- [67] A. Grignard, P. Taillandier, B. Gaudou, D. A. Vo, N. Q. Huynh, and A. Drogoul. "GAMA 1.6: advancing the art of complex agent-based modeling and simulation." In: *PRIMA 2013: Principles and Practice of Multi-Agent Systems*. Ed. by G. Boella, E. Elkind, B. T. R. Savarimuthu, F. Dignum, and M. K. Purvis. 2013, pp. 117–131. doi: 10.1007/978-3-642-44927-7_9.
- [68] V. Grimm and S. F. Railsback. *Individual-Based Modeling and Ecology*. Princeton University Press, 2013.
- [69] P. Grubel, H. Kaiser, J. Cook, and A. Serio. "The performance implication of task size for applications on the HPX runtime system." In: *2015 IEEE International Conference on Cluster Computing*. 2015, pp. 682–689. doi: 10.1109/cluster.2015.119.
- [70] Q. Guan and K. C. Clarke. "A general-purpose parallel raster processing programming library test application using a geographic cellular automata model." In: *International Journal of Geographical Information Science* 24.5 (2010), pp. 695–722. doi: 10.1080/13658810902984228.
- [71] N. Guarino, D. Oberle, and S. Staab. "What is an ontology?" In: *Handbook on Ontologies*. Ed. by S. Staab and R. Studer. 2009, pp. 1–17. doi: 10.1007/978-3-540-92673-3_0.
- [72] A. A. Hagberg, D. A. Schult, and P. J. Swart. "Exploring network structure, dynamics, and function using NetworkX." In: *Proceedings of the 7th Python in Science Conference*. 2008, pp. 11–15.
- [73] T. Heller, H. Kaiser, and K. Iglberger. "Application of the ParalleX execution model to stencil-based problems." In: *Computer Science – Research and Development* 28 (2013), pp. 253–261. doi: 10.1007/s00450-012-0217-1.
- [74] T. Heller, B. Adelstein Lelbach, K. Huck, J. Biddiscombe, P. Grubel, A. Koniges, M. Kretz, D. Marcello, D. Pfander, A. Serio, J. Frank, G. Clayton, D. Pflüger, D. Eder, and H. Kaiser. "Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger of two stars." In: *The International Journal of High Performance Computing Applications* 33.4 (2019), pp. 699–715. doi: 10.1177/1094342018819744.

- [75] T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, and H. Kaiser. "HPX – An open source C++ standard library for parallelism and concurrency." In: *Proceedings of OpenSuCo 2017, Denver, Colorado USA, November 2017*. 2017, p. 5.
- [76] M. Hendriks. *Physical Hydrology*. 1st ed. Oxford University Press, 2010.
- [77] K. Hinsen. "Caring for your data." In: *Computing in Science & Engineering* 14.6 (2012), pp. 70–74. doi: 10.1109/mcse.2012.108.
- [78] G. Hoek, R. Beelen, K. de Hoogh, D. Vienneau, J. Gulliver, P. Fischer, and D. Briggs. "A review of land-use regression models to assess spatial variation of outdoor air pollution." In: *Atmospheric Environment* 42.33 (2008), pp. 7561–7578. doi: 10.1016/j.atmosenv.2008.05.057.
- [79] N. Holst and G. F. Belete. "Domain-specific languages for ecological modelling." In: *Ecological Informatics* 27 (2015), pp. 26–38. doi: 10.1016/j.ecoinf.2015.02.005.
- [80] E. Holzbecher. *Environmental Modeling. Using MATLAB*. 2nd ed. Springer-Verlag Berlin-Heidelberg, 2012. doi: 10.1007/978-3-642-22042-5.
- [81] K. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. Malony, T. Sterling, and R. Fowler. "An autonomic performance environment for exascale." In: *Supercomputing Frontiers and Innovations* 2.3 (2015). doi: 10.14529/jsfi150305.
- [82] International Organization for Standardization. *Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – System and software quality models*. Standard. 2011.
- [83] W. Jakob, J. Rhineland, and D. Moldovan. *pybind11 – Seamless operability between C++11 and Python*. URL: <https://github.com/pybind/pybind11> (visited on 07/12/2019).
- [84] A. Jjumba and S. Dragicevic. "Integrating GIS-based Geo-Atom theory and voxel automata to simulate the dispersal of airborne pollutants." In: *Transactions in GIS* 19.4 (2014), pp. 582–603. doi: 10.1111/tgis.12113.
- [85] K. M. Johnston. *Agent Analyst: Agent-Based Modeling in ArcGIS*. Esri Press, Redlands, California, 2013.

- [86] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open source scientific tools for Python*. URL: <https://www.scipy.org> (visited on 07/12/2019).
- [87] K. de Jong and D. Karssenberg. "A physical data model for spatio-temporal objects." In: *Environmental Modelling & Software* 122 (2019), p. 104553. DOI: 10.1016/j.envsoft.2019.104553.
- [88] K. de Jong, D. Panja, M. van Kreveld, and D. Karssenberg. "An environmental modelling framework based on asynchronous many-tasks: scalability and usability." In: *Environmental Modelling & Software* 139 (2021), p. 104998. DOI: 10.1016/j.envsoft.2021.104998.
- [89] K. de Jong and O. Schmitz. *computationalgeography/lue: LUE-o.3.0: Scientific database and environmental modelling framework*. Version 0.3.0. Sept. 2021. DOI: 10.5281/zenodo.5535686.
- [90] H. Kaiser, M. Brodowicz, and T. Sterling. "ParalleX: An advanced parallel execution model for scaling-impaired applications." In: *2009 International Conference on Parallel Processing Workshops*. 2009, pp. 394–401. DOI: 10.1109/icppw.2009.14.
- [91] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Bergé, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang. "HPX - The C++ standard library for parallelism and concurrency." In: *Journal of Open Source Software* 5.53 (2020), p. 2352. DOI: 10.21105/joss.02352.
- [92] H. Kaiser, B. A. Lelbach, M. Simberg, T. Heller, A. Bergé, J. Biddiscombe, A. R., A. Bikineev, G. Mercer, A. Schäfer, K. Huck, A. S. Lemoine, T. Kwon, J. Habraken, M. Anderson, M. Copik, S. R. Brandt, M. Stumpf, D. Bourgeois, D. Blank, rstobaugh, S. Jakobovits, V. Amatya, L. Viklund, N. Gupta, P. Diehl, Z. Khatami, D. Bacharwar, and S. Y. Tapasweni Pathak. *STELLAR-GROUP/hpx: HPX V1.5.0: The C++ standards library for parallelism and concurrency*. Version 1.5.0. Sept. 2020. DOI: 10.5281/zenodo.4011590.
- [93] H. Kaiser, M. Simberg, B. A. Lelbach, T. Heller, A. Bergé, J. Biddiscombe, A. R., A. Bikineev, G. Mercer, A. Schäfer, K. Huck, A. S. Lemoine, T. Kwon, J. Habraken, A. Nair, M. Anderson, S. R.

- Brandt, M. Copik, srinivasyadav18, Finomnis, D. Bourgeois, D. Blank, G. Gonidelis, N. Gupta, rstobaugh, S. Jakobovits, V. Amatya, L. Viklund, P. Diehl, and Z. Khatami. *STELLAR-GROUP/hpx: HPX V1.7.1: The C++ standards library for parallelism and concurrency*. Version 1.7.1. Aug. 2021. DOI: 10.5281/zenodo.5185328.
- [94] L. V. Kale and S. Krishnan. "CHARM++: a portable concurrent object oriented system based on C++." In: *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. 1993, pp. 91–108. DOI: 10.1145/165854.165874.
- [95] D. Karssenberg and K. de Jong. "Dynamic environmental modelling in GIS: 1. Modelling in three spatial dimensions." In: *International Journal of Geographical Information Science* 19.5 (2005), pp. 559–579. DOI: 10.1080/13658810500032362.
- [96] D. Karssenberg. "Building Dynamic Spatial Environmental Models." PhD thesis. the Netherlands: Utrecht University, 2002. ISBN: 90-6809-341-X.
- [97] D. Karssenberg. "The value of environmental modelling languages for building distributed hydrological models." In: *Hydrological Processes* 16.14 (2002), pp. 2751–2766. DOI: 10.1002/hyp.1068.
- [98] D. Karssenberg. "Upscaling of saturated conductivity for Hortonian runoff modelling." In: *Advances in Water Resources* 29.5 (2006), pp. 735–759. DOI: 10.1016/j.advwatres.2005.06.012.
- [99] D. Karssenberg and M. F. P. Bierkens. "Early-warning signals (potentially) reduce uncertainty in forecasted timing of critical shifts." In: *Ecosphere* 3.2 (2012). DOI: 10.1890/es11-00293.1.
- [100] D. Karssenberg and J. S. Bridge. "A three-dimensional numerical model of sediment transport, erosion and deposition within a network of channel belts, floodplain and hill slope: extrinsic and intrinsic controls on floodplain dynamics and alluvial architecture." In: *Sedimentology* 55 (2008), pp. 1717–1745. DOI: 10.1111/j.1365-3091.2008.00965.x.
- [101] D. Karssenberg, O. Schmitz, P. Salamon, K. de Jong, and M. F. Bierkens. "A software framework for construction of process-based stochastic spatio-temporal models and data as-

- simulation." In: *Environmental Modelling & Software* 25.4 (2010), pp. 489–502. doi: 10.1016/j.envsoft.2009.10.004.
- [102] J. Kazil, D. Masad, and A. Crooks. "Utilizing Python for agent-based modeling: the Mesa framework." In: *Social, Cultural, and Behavioral Modeling*. Ed. by R. Thomson, H. Bisgin, C. Dancy, A. Hyder, and M. Hussain. 2020, pp. 308–317. doi: 10.1007/978-3-030-61255-9_30.
- [103] Z. Khatami, H. Kaiser, P. Grubel, A. Serio, and J. Ramanujam. "A massively parallel distributed N-body application implemented with HPX." In: *7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 2016, pp. 57–64. doi: 10.1109/scala.2016.012.
- [104] K. Kjenstad. "On the integration of object-based models and field-based models in GIS." In: *International Journal of Geographical Information Science* 20.5 (2006), pp. 491–509. doi: 10.1080/13658810600607329.
- [105] L. Klein and A. Taaheri. *HDF-EOS5 Data Model, File Format and Library*. Tech. rep. 2007, pp. 1–56.
- [106] M. Kocifaj and L. Kómar. "Modeling diffuse irradiance under arbitrary and homogeneous skies: comparison and validation." In: *Applied Energy* 166 (2016), pp. 117–127. doi: 10.1016/j.apenergy.2016.01.024.
- [107] B. Kotyra, Ł. Chabudziński, and P. Stpiczyński. "High-performance parallel implementations of flow accumulation algorithms for multicore architectures." In: *Computers & Geosciences* 151 (2021), p. 104741. doi: 10.1016/j.cageo.2021.104741.
- [108] W. Kuhn. "Core concepts of spatial information for transdisciplinary research." In: *International Journal of Geographical Information Science* 26.12 (2012), pp. 2267–2276. doi: 10.1080/13658816.2012.722637.
- [109] W. Kuhn. "What is field and object information?" In: *Extended Abstract Proceedings of the GIScience 2014*. Ed. by K. Stewart, E. Pebesma, G. Navratil, P. Fogliaroni, M. Duckham, and A. U. Frank. 2014, pp. 96–98.
- [110] N. Lana-Renault and D. Karssenber. "PyCatch: Component based hydrological catchment modelling." In: *Cuadernos de Investigación Geográfica* 39.2 (2013), pp. 315–333. doi: 10.18172/cig.1993.

- [111] X. Li, J. Yang, X. Guan, and H. Wu. "An event-driven spatiotemporal data model (E-ST) supporting dynamic expression and simulation of geographic processes." In: *Transactions in GIS* 18.2012 (2014). DOI: 10.1111/tgis.12127.
- [112] Z. Li, F. Wang, X. Zheng, W. Jiang, Q. Meng, and B. Liu. "GIS based dynamic modeling of fire spread with heterogeneous cellular automation model and standardized emergency management protocol." In: *Proceedings of the 3rd ACM SIGSPATIAL Workshop on Emergency Management Using*. 2017. DOI: 10.1145/3152465.3152470.
- [113] Y. Liu, M. F. Goodchild, Q. Guo, Y. Tian, and L. Wu. "Towards a general field model and its order in GIS." In: *International Journal of Geographical Information Science* 22.6 (2008), pp. 623–643. DOI: 10.1080/13658810701587727.
- [114] P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind. *Geographic Information Systems & Science*. 3rd ed. John Wiley & Sons, 2011.
- [115] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. "MASON: A multiagent simulation environment." In: *Simulation* 81.7 (2005), pp. 517–527. DOI: 10.1177/0037549705058073.
- [116] MathWorks. *MATLAB*. URL: <https://www.mathworks.com> (visited on 12/15/2021).
- [117] J. Mennis. "Multidimensional map algebra: design and implementation of a spatio-temporal GIS processing language." In: *Transactions in GIS* 14.1 (2010), pp. 1–21. DOI: 10.1111/j.1467-9671.2009.01179.x.
- [118] J. Mennis, R. Viger, and C. D. Tomlin. "Cubic map algebra functions for spatio-temporal analysis." In: *Cartography and Geographic Information Science* 32.1 (2005), pp. 17–32. DOI: 10.1559/1523040053270765.
- [119] M. Mernik, J. Heering, and A. M. Sloane. "When and how to develop domain-specific languages." In: *ACM Computing Surveys* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.
- [120] S. H. Mernild, N. T. Knudsen, J. C. Yde, and J. K. Malmros. "Detailed spatiotemporal albedo observations at Greenland's Mittivakkat Gletscher." In: *EGU General Assembly Conference Abstracts*. Vol. 17. 2015, p. 2643.

- [121] F. Molteni, R. Buizza, T. N. Palmer, and T. Petroliaigis. "The ECMWF ensemble prediction system: methodology and validation." In: *Quarterly Journal of the Royal Meteorological Society* 122.529 (1996), pp. 73–119. DOI: 10.1002/qj.49712252905.
- [122] N. Moreno, A. Ménard, and D. J. Marceau. "VecGCA: A vector-based geographic cellular automata model allowing geometric transformations of objects." In: *Environment and Planning B: Planning and Design* 35.4 (2008), pp. 647–665. DOI: 10.1068/b33093.
- [123] MPI Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015. URL: <https://www.mpi-forum.org>.
- [124] S. Namany, R. Govindan, L. Alfagih, G. McKay, and T. Al-Ansari. "Sustainable food security decision-making: an agent-based modelling approach." In: *Journal of Cleaner Production* 255 (2020), p. 120296. DOI: 10.1016/j.jclepro.2020.120296.
- [125] M. Neteler, M. Bowman, M. Landa, and M. Metz. "GRASS GIS: A multi-purpose open source GIS." In: *Environmental Modelling & Software* 31 (2012), pp. 124–130. DOI: 10.1016/j.envsoft.2011.11.014.
- [126] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko. "Complex adaptive systems modeling with Repast Symphony." In: *Complex Adaptive Systems Modeling* 1.1 (2013), p. 3. DOI: 10.1186/2194-3206-1-3.
- [127] I. Noy-Meir. "Stability of grazing systems: an application of predator-prey graphs." In: *Journal of Ecology* 63.2 (1975), pp. 459–481.
- [128] J. F. O'Callaghan and D. M. Mark. "The extraction of drainage networks from digital elevation data." In: *Computer Vision, Graphics, and Image Processing* 28.3 (1984), pp. 323–344. DOI: 10.1016/s0734-189x(84)80011-0.
- [129] T. E. Oliphant. *A Guide to NumPy*. USA: Trelgol Publishing, 2006.
- [130] Open Geospatial Consortium. *The OpenGIS Abstract Specification Topic 8: Relationships Between Features*. Tech. rep. 1999. URL: <https://www.opengeospatial.org/standards/as>.

- [131] Open Geospatial Consortium. *The OpenGIS Abstract Specification Topic 6: Schema for coverage geometry and functions*. Tech. rep. 2006. URL: <https://www.opengeospatial.org/standards/as>.
- [132] Open Geospatial Consortium. *The OpenGIS Abstract Specification Topic 5: Features*. Tech. rep. 2009. URL: <https://www.opengeospatial.org/standards/as>.
- [133] Open Geospatial Consortium. *OGC Reference Model*. Tech. rep. 2011. URL: <https://www.opengeospatial.org/standards>.
- [134] Open Geospatial Consortium. *OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture*. Tech. rep. 2011. URL: <https://www.opengeospatial.org/standards/sfa>.
- [135] L. Ortega and A. Rueda. "Parallel drainage network computation on CUDA." In: *Computers & Geosciences* 36.2 (2010), pp. 171–178. doi: 10.1016/j.cageo.2009.07.005.
- [136] P. Panagos, P. Borrelli, J. Poesen, C. Ballabio, E. Lugato, K. Meusburger, L. Montanarella, and C. Alewell. "The new assessment of soil loss by water erosion in Europe." In: *Environmental Science & Policy* 54 (2015), pp. 438–447. doi: 10.1016/j.envsci.2015.08.012.
- [137] D. C. Parker. "Integration of geographic information systems and agent-based models of land use: prospects and challenges." In: *GIS, Spatial Analysis and Modeling*. Ed. by D. Maguire, M. Goodchild, and M. Batty. ESRI press, 2005, pp. 403–422.
- [138] D. C. Parker, S. M. Manson, M. A. Janssen, M. J. Hoffmann, and P. Deadman. "Multi-agent systems for the simulation of land-use and land-cover change: a review." In: *Annals of the Association of American Geographers* 93.2 (2003), pp. 314–337. doi: 10.1111/1467-8306.9302004.
- [139] D. C. Parker and T. Filatova. "A conceptual design for a bilateral agent-based land market with heterogeneous economic agents." In: *Computers, Environment and Urban Systems* 32.6 (2008), pp. 454–463. doi: 10.1016/j.compenvurbsys.2008.09.012.
- [140] D. Patterson and J. Hennessy. *Computer Organization Design*. 4th ed. Morgan Kaufmann, 2008.

- [141] T. P. Peixoto. *The graph-tool Python library*. 2017. DOI: 10.6084/m9.figshare.1164194.
- [142] D. J. Peuquet. "Making space for time: issues in space-time data representation." In: *GeoInformatica* 5.1 (2001), pp. 11–32.
- [143] PostGIS development community. *PostGIS – Spatial and Geographic Objects for PostgreSQL*. URL: <https://postgis.net> (visited on 07/12/2019).
- [144] D. Pullar. "MapScript: A map algebra programming language incorporating neighborhood analysis." In: *GeoInformatica* 5.1 (2001), pp. 145–163.
- [145] E. Pultar, T. J. Cova, M. Yuan, and M. F. Goodchild. "EDGIS: A dynamic GIS based on space time points." In: *International Journal of Geographical Information Science* 24.3 (2010), pp. 329–346. DOI: 10.1080/13658810802644567.
- [146] C.-Z. Qin, L.-J. Zhan, A.-X. Zhu, and C.-H. Zhou. "A strategy for raster-based geocomputation under different parallel computing platforms." In: *International Journal of Geographical Information Science* 28.11 (2014), pp. 2127–2144. DOI: 10.1080/13658816.2014.911300.
- [147] R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2013. URL: <https://www.r-project.org>.
- [148] S. F. Railsback and V. Grimm. *Agent-Based and Individual-Based modeling*. 2nd ed. Princeton University Press, 2019.
- [149] R. J. Rockett, A. Arnott, C. Lam, R. Sadsad, V. Timms, K.-A. Gray, J.-S. Eden, S. Chang, M. Gall, J. Draper, E. M. Sim, N. L. Bachmann, I. Carter, K. Basile, R. Byun, M. V. O'Sullivan, S. C.-A. Chen, S. Maddocks, T. C. Sorrell, D. E. Dwyer, E. C. Holmes, J. Kok, M. Prokopenko, and V. Sintchenko. "Revealing COVID-19 transmission in Australia by SARS-CoV-2 genome sequencing and agent-based modeling." In: *Nature Medicine* 26.9 (2020), pp. 1398–1404. DOI: 10.1038/s41591-020-1000-7.
- [150] H. Sbihi, R. W. Allen, A. Becker, J. R. Brook, P. Mandhane, J. A. Scott, M. R. Sears, P. Subbarao, T. K. Takaro, S. E. Turvey, and M. Brauer. "Perinatal exposure to traffic-related air pollution and atopy at 1 year of age in a multi-center Canadian birth cohort study." In: *Environmental Health Perspectives* 123.9 (2015), pp. 902–908. DOI: 10.1289/ehp.1408700.

- [151] M. J. Schelhaas, K. Kramer, H. Peltola, D. C. van der Werf, and S. M. J. Wijdeven. "Introducing tree interactions in wind damage simulation." In: *Ecological Modelling* 207.2-4 (2007), pp. 197–209. doi: 10.1016/j.ecolmodel.2007.04.025.
- [152] P. Schippers, A. J. A. van Teeffelen, J. Verboom, C. C. Vos, K. Kramer, and M. F. WallisDeVries. "The impact of large herbivores on woodland-grassland dynamics in fragmented landscapes: the role of spatial configuration and disturbance." In: *Ecological Complexity* 17 (2014), pp. 20–31. doi: 10.1016/j.ecocom.2013.07.002.
- [153] O. Schmitz. *Campo environmental modelling framework*. URL: <https://campo.computationalgeography.org> (visited on 12/12/2021).
- [154] J. A. Sharp. "Data oriented program design." In: *ACM SIG-PLAN Notices* 15.9 (1980), pp. 44–57. doi: 10.1145/947706.947713.
- [155] E. Shook, M. E. Hodgson, S. Wang, B. Behzad, K. Soltani, A. Hiscox, and J. Ajayakumar. "Parallel cartographic modeling: a methodology for parallelizing spatial data processing." In: *International Journal of Geographical Information Science* 30.12 (2016), pp. 2355–2376. doi: 10.1080/13658816.2016.1172714.
- [156] SQLite developers. *SQLite*. URL: <https://www.sqlite.org> (visited on 07/12/2019).
- [157] J. Sten, H. Lilja, J. Hyväluoma, J. Westerholm, and M. Aspnäs. "Parallel flow accumulation algorithms for graphical processing units with application to RUSLE model." In: *Computers & Geosciences* 89 (2016), pp. 88–95. doi: 10.1016/j.cageo.2016.01.006.
- [158] K. Sullivan, M. Coletti, and S. Luke. *GeoMason: GeoSpatial Support for MASON*. Tech. rep. Department of Computer Science, George Mason University, 2010.
- [159] E. Sutanudjaja, L. van Beek, S. de Jong, F. van Geer, and M. Bierkens. "Large-scale groundwater modeling using global datasets: a test case for the Rhine-Meuse basin." In: *Hydrology and Earth System Sciences* 15 (2011), pp. 2913–2935. doi: 10.5194/hess-15-2913-2011.

- [160] E. H. Sutanudjaja, R. van Beek, N. Wanders, Y. Wada, J. H. C. Bosmans, N. Drost, R. J. van der Ent, I. E. M. de Graaf, J. M. Hoch, K. de Jong, D. Karssenbergh, P. L. López, S. Peßenteiner, O. Schmitz, M. W. Straatsma, E. Vannamettee, D. Wisser, and M. F. P. Bierkens. "PCR-GLOBWB 2: A 5 arc-minute global hydrological and water resources model." In: *Geoscientific Model Development Discussions* (2017), pp. 1–41. DOI: 10.5194/gmd-2017-288.
- [161] V. Te Chow, D. Maidment, and L. Mays. *Applied Hydrology*. Civil Engineering. McGraw-Hill, 1988.
- [162] The HDF Group. *Hierarchical Data Format, version 5*. URL: <https://www.hdfgroup.org/solutions/hdf5> (visited on 09/23/2021).
- [163] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos. "A taxonomy of task-based parallel programming technologies for high-performance computing." In: *The Journal of Supercomputing* 74 (2018), pp. 1422–1434. DOI: 10.1007/s11227-018-2238-4.
- [164] K. Tomaž, O. Nuno, M. Marjan, P. V. J. Maria, Č. Matej, D. C. Daniela, and H. R. Pedro. "Comparing general-purpose and domain-specific languages: an empirical study." In: *Computer Science and Information Systems* 7.2 (2010), pp. 247–264. DOI: 10.2298/csis1002247k.
- [165] C. Tomlin. "A map algebra." In: *Proceedings of Harvard Computer Conference*. Cambridge, MA, 1983.
- [166] D. Tomlin. *Geographic Information Systems and Cartographic Modeling*. 1st ed. Prentice-Hall, 1990.
- [167] A. Trucchia, M. D'Andrea, F. Baghino, P. Fiorucci, L. Ferraris, D. Negro, A. Gollini, and M. Severino. "PROPAGATOR: An operational cellular-automata based wildfire simulator." In: *Fire* 3.3 (2020). DOI: 10.3390/fire3030026.
- [168] UNIDATA. *Network Common Data Form, version 4*. URL: <https://www.unidata.ucar.edu/software/netcdf/docs/index.html> (visited on 07/12/2019).

- [169] H. T. Valentine and A. Mäkelä. "Bridging process-based and empirical approaches to modeling tree growth." In: *Tree Physiology* 25.7 (2005), pp. 769–779. DOI: 10.1093/treephys/25.7.769.
- [170] V. Voudouris. "Towards a unifying formalisation of geographic representation: the object field model with uncertainty and semantics." In: *International Journal of Geographical Information Science* 24.12 (2010), pp. 1811–1828. DOI: 10.1080/13658816.2010.488237.
- [171] H. Wang, A. Ghosh, J. Ding, R. Sarkar, and J. Gao. "Heterogeneous interventions reduce the spread of COVID-19 in simulations on real mobility data." In: *Scientific Reports* 11 (2021). DOI: 10.1038/s41598-021-87034-z.
- [172] X. Wang and D. Pullar. "Describing dynamic modeling for landscapes with vector map algebra in GIS." In: *Computers & Geosciences* 31.8 (2005), pp. 956–967. DOI: 10.1016/j.cageo.2005.02.015.
- [173] U. Wilensky. *NetLogo*. URL: <https://ccl.northwestern.edu/netlogo>.
- [174] S. Winter. "Bridging vector and raster representation in GIS." In: *Proceedings of the 6th ACM International symposium on Advances in geographic information systems*. 1998, pp. 57–62.
- [175] S. Winter and S. Nittel. "Formal information modelling for standardisation in the spatial domain." In: *International Journal of Geographical Information Science* 17.8 (2003), pp. 721–741. DOI: 10.1080/13658810310001596067.
- [176] D. Yamazaki, D. Ikeshima, J. Sosa, P. D. Bates, G. H. Allen, and T. M. Pavelsky. "MERIT Hydro: A high-resolution global hydrography map based on latest topography dataset." In: *Water Resources Research* 55.6 (2019), pp. 5053–5073. DOI: 10.1029/2019wr024873.
- [177] M. Yuan. "Representing complex geographic phenomena in GIS." In: *Cartography and Geographic Information Science* 28.2 (2001), pp. 83–96. DOI: 10.1559/152304001782173718.
- [178] B. P. Zeigler, A. Muzy, and E. Kofman. *Theory of Modeling and Simulation*. 3rd ed. Elsevier, 2019. DOI: 10.1016/c2016-0-03987-6.

- [179] G. Zhou, H. Wei, and S. Fu. "A fast and simple algorithm for calculating flow accumulation matrices from raster digital elevation." In: *Frontiers of Earth Science* 13 (2019), pp. 317–326. doi: 10.1007/s11707-018-0725-9.
- [180] G. Ziervogel, M. Bithell, R. Washington, and T. Downing. "Agent-based social simulation: a method for assessing the impact of seasonal climate forecast applications among small-holder farmers." In: *Agricultural Systems* 83.1 (2005), pp. 1–26. doi: 10.1016/j.agsy.2004.02.009.
- [181] K. Zigner, L. M. V. Carvalho, S. Peterson, F. Fujioka, G.-J. Duine, C. Jones, D. Roberts, and M. Moritz. "Evaluating the ability of FARSITE to simulate wildfires influenced by extreme, downslope winds in Santa Barbara, California." In: *Fire* 3.3 (2020), p. 29. doi: 10.3390/fire3030029.

SUMMARY

Simulating the state and processes that are part of a real-world environmental system using a computer model can be a useful way to help better understand how such a system works, or to make predictions of the state of such a system at unvisited locations in time or space, or under modified conditions. Examples of processes that are simulated using computer models are meteorological processes to predict tomorrow's weather, hydrological processes to predict floods, transmission processes to understand the spread of a virus, and crowd dynamics to improve the design of a new train station.

The focus of the research presented in this thesis is on simulation modelling of geographical systems, in which the state of such a system is simulated forward through time. In a simulation model, the state of a system is represented by a set of state variables, and the processes by a set of modelling operations. During a simulation, the modelling operations update the state variables, simulating the change of the system's state over time. Depending on the characteristic of the real-world phenomena which the state variables represent, some state is most conveniently represented by discrete objects, and other by continuous fields. For example, the properties of individual trees can be represented by objects, but the distribution of biomass in a forest may be better represented by a continuous field, depending on the data availability and the purpose of the modelling study.

Developing a non-trivial simulation model takes a lot of time, and involves developing software. Modelling frameworks exist that contain pre-built building blocks that make it easier for model developers to build their models. They hide much of the complexity involved in implementing the data structures and algorithms required to represent state variables and modelling operations.

In the domain of geographical simulation modelling, two major approaches are used for developing simulation models using a modelling framework: agent-based modelling and field-based modelling. In agent-based modelling, state variables are represented by discrete objects and modelling operations manipulate these. Continuous fields can be used, but mostly as a way for the objects to sense the environment. In general, it is not possible to develop advanced field-based

models using an agent-based modelling framework. On the other hand, in field-based modelling state variables are represented by continuous fields and modelling operations manipulate these. It is not possible to develop advanced agent-based models using a field-based modelling approach.

The overall goal behind the research presented in this thesis is to gain knowledge required to be able to develop a modelling framework for simulating large spatio-temporal systems of both agents and fields. That knowledge is especially relevant for developers of modelling frameworks. The objectives focused on are related to integrating the different representations used in agent- and field-based modelling, and to improving the scalability of models created with a modelling framework.

REPRESENTING GEOGRAPHICAL STATE

The problem that is the focus of the first part of this thesis is that, to be able to represent the state of a geographical system, it is often necessary to be able to represent some of it using discrete objects, and some of it using continuous fields. The current generation of modeling frameworks is not capable of allowing the model developer to conveniently create simulation models in which this is the case.

The approach taken to improve this situation is inspired by the map algebra approach used in some field-based modelling frameworks. This is a convenient approach for implementing models because the syntax of model expressions is similar to that of mathematical expressions, with which model developers are already familiar. The high level of abstraction is a result of abstracting away the differences between spatial rasters and scalars, and by implicitly iterating over all cells within modelling operations. As a first step towards a modelling framework supporting the integrated simulation of both agents and fields, a conceptual data model is presented, with which the differences between discrete objects and continuous fields are abstracted away. Many kinds of information related to objects, like their location in simulated time and space, their properties, and their relations can be represented by it. Fields can be represented as well, as a spatial property of a “research area” object. The conceptual data model can serve as a blueprint for a type for state variables in a new modelling framework and as such, is something model developers will interact with when developing their models. A future modelling framework

can support operations that, besides iterating over any raster cells, also iterates over all objects. That would keep the syntax of simulation models similar to current field-based models, but would allow integrating agent- and field-based modelling.

In addition to the conceptual data model, a physical data model is presented with which state variables, organized according to the conceptual data model, can be stored in a data set. The actual information referred to by state variables can be very different, depending on several aspects. For example, the contents of some collections of objects may vary through time, in case those objects can be “born” and “die”, or not. Also, objects may be mobile or stationary, they may have zero or more properties, some of which may be variable through time or constant, etc. Three layers of abstractions are defined with which this diverse information about objects can be organized in a dataset. With the lowest level of abstraction, six general kinds of arrays of temporal information can be stored, and with the upper level, the actual spatio-temporal objects the state variables refer to can be stored. The physical data model is implemented as a library, providing C++ and Python APIs on top of the upper level of abstraction, and can be used to performing I/O to an HDF5 dataset.

SCALABLE COMPUTING

Simulation models often become larger over time, either due to an increase in the size of the state variables or an increase of the number of modelling operations, or both. This happens, for example, because a new satellite has been taken into production, providing more detailed information about the state of the environment. In order to still be able to execute models that become larger, these models must be able to efficiently use more hardware. Otherwise they may take too long to execute, or may fail to execute at all.

Simulation models that scale well are able to use additional hardware efficiently. For a model developed using a modelling framework to scale well, the modelling framework must scale well. This implies that its data structures and algorithms must support the development of scalable models, whatever the combination of state variables and modelling operations.

An approach for developing scalable software which is especially promising in the context of modelling frameworks is called asynchronous many-tasks (AMT). When using AMT the software devel-

oper can define tasks that perform some work and the dependencies between these tasks. The AMT runtime is responsible for making sure these tasks get executed as fast as possible using all available hardware, taking their interdependencies into account. Some of the potential advantages of AMT in the context of a modelling framework is that there tends to be fewer synchronization points in the code, which limit the scalability, and that concurrent tasks from multiple modelling operations can be scheduled for execution in parallel, hiding some of the load imbalance that may be present.

The problem that is the focus of the second part of this thesis is that of how the AMT approach can be used in the design and implementation of a modelling framework, given the requirement that models developed with it perform and scale well.

To represent continuous fields in models, a partitioned array data structure is designed and implemented. To represent processes, a set of local, focal, zonal, global, and flow routing operations is designed and implemented that accept partitioned arrays as arguments and return new partitioned arrays. A useful feature of the use of AMT is that modelling operations execute faster than the work that is asynchronously spawned by them. This makes it possible for concurrent work from subsequent modelling operations to execute in parallel.

The flow routing algorithms are implemented in terms of tasks that have the responsibility of calculating a result for a single partition. These tasks only depend on tasks handling directly neighbouring partitions, and information can be exchanged between them. Tasks handling partitions containing upstream parts of large scale streams finish before those handling partitions containing downstream parts of large scale streams, and the corresponding result partitions can already participate in any subsequent modelling operations.

Experimental results show that the performance and scalability of models created with the new framework is good. Additional hardware can be used efficiently. Additionally, an experiment executing two flow routing operations showed that load imbalance of the first call was hidden by work spawned by the second operation.

The partitioned array data structure and the modelling operations were implemented in C++ and used the HPX AMT library and runtime. Model developers can use the Python language to define their models. These models can execute unmodified on laptops and on computer clusters.

INTEGRATED AGENT- AND FIELD-BASED MODELLING

The conceptual and physical data models for representing various kinds of state variables used in agent- and field-based modelling can be used as a basis for representing state variables of an integrated modelling framework with which both discrete objects and continuous fields can be manipulated. The AMT approach used in the implementation of a prototype modelling framework containing a partitioned array data structure and a set of local, focal, zonal, global and flow routing operations proves to be a good approach. Model developers can develop large models with it that are capable of using additional hardware efficiently, either to get model results sooner or to be able to run the model on larger data sets.

Several topics remain to be researched. An important one is how to design a model development interface that is easy to use by model developers, hiding all complexity that is unrelated to defining model state and processes, and with which integrated agent- and field-based models can be developed. Another topic is how to use AMT in a modelling framework for scalable agent-based modelling and integrated agent- and field-based modelling.

SAMENVATTING

Ruimtelijke processen in het landschap kunnen worden nagebootst met behulp van een computermodel. Dit kan om meerdere redenen waardevol zijn. Ten eerste kan een computermodel helpen om beter te begrijpen hoe het ruimtelijke systeem werkt. Ten tweede kan het worden gebruikt om voorspellingen te maken voor locaties of momenten in de tijd waarvoor geen metingen beschikbaar zijn. Tot slot kan het worden gebruikt om scenarios van gewijzigde condities door te rekenen. Voorbeelden van processen die worden nagebootst met computermodellen zijn meteorologische processen om het weer van morgen te voorspellen, hydrologische processen om overstromingen te voorspellen, overdrachtsprocessen om te begrijpen hoe een virus zich verspreidt en “crowd dynamics” om het gedrag van menigtes te simuleren om het ontwerp van een nieuw treinstation te verbeteren.

De focus van het onderzoek dat in dit proefschrift wordt beschreven ligt op het modelmatig nabootsen van geografische systemen, waarin de staat van het systeem voorwaarts door de tijd wordt gesimuleerd. In een simulatiemodel wordt de staat van een systeem gerepresenteerd door de set van toestandvariabelen, en de processen door een set van modelleeroperaties. Tijdens een simulatie verversen de modelleeroperaties de toestandvariabelen, waarmee de verandering van de staat door de tijd wordt nagebootst. Afhankelijk van de eigenschappen van de fenomenen uit de echte wereld die door de toestandvariabelen worden gerepresenteerd, kan de ene staat beter worden gerepresenteerd door discrete objecten en andere beter door continue velden. De eigenschappen van individuele bomen kunnen bijvoorbeeld worden gerepresenteerd door objecten, maar de verdeling van biomassa in een bos kan mogelijk beter worden gerepresenteerd door een continu veld, afhankelijk van de beschikbaarheid van gegevens en het doel van de modelleerstudie.

Het ontwikkelen van simulatiemodellen kost veel tijd en omvat het ontwikkelen van software. Er bestaan modelleerraamwerken met daarin bouwstenen die het voor modelontwikkelaars makkelijker maken om hun modellen te maken. Deze verbergen veel van de complexiteit die betrekking heeft op het implementeren van de datastructuren

en algoritmes die nodig zijn om de toestandvariabelen en modelleeroperaties te representeren.

Binnen de geografie en verwante domeinen worden met name twee aanpakken gebruikt voor het ontwikkelen van simulatiemodellen met modelleerraamwerken: agent-gebaseerd modelleren en veld-gebaseerd modelleren. Met agent-gebaseerd modelleren worden toestandvariabelen gerepresenteerd door discrete objecten, die door modelleeroperaties worden gemanipuleerd. Continue velden kunnen wel worden gebruikt, maar met name als een manier om de omgeving van de agenten te representeren. In het algemeen is het niet mogelijk om geavanceerde veld-gebaseerde modellen te ontwikkelen met een agent-gebaseerd modelleerraamwerk. Andersom worden met veld-gebaseerd modelleren toestandvariabelen gerepresenteerd door middel van continue velden, die door modelleeroperaties worden gemanipuleerd. Het is niet mogelijk om geavanceerde agent-gebaseerde modellen te ontwikkelen met een veld-gebaseerd modelleerraamwerk.

Het achterliggende doel van het onderzoek dat in dit proefschrift wordt beschreven is om kennis te verkrijgen die nodig is om een modelleerraamwerk te kunnen ontwikkelen voor het simuleren van grote ruimtelijk-temporele systemen van agenten en velden. Die kennis is met name relevant voor ontwikkelaars van modelleerraamwerken. De doelen waarop gefocust wordt zijn gerelateerd aan de integratie van de verschillende representaties die worden gebruikt in agent- en veld-gebaseerd modelleren, en aan het verbeteren van de schaalbaarheid van modellen die met een modelleerraamwerk worden gemaakt.

HET REPRESENTEREN VAN GEOGRAFISCHE STAAT

Het probleem waar in het eerste deel van deze thesis op wordt gefocust is dat, om de staat van een geografisch systeem te representeren, het vaak nodig is om een deel ervan te representeren door middel van discrete objecten, en een deel door middel van continue velden. De huidige generatie modelleerraamwerken stelt de modelontwikkelaar niet in staat om op een handige manier simulatiemodellen te maken waarvoor dit geldt.

De aanpak waarvoor gekozen is om deze situatie te verbeteren is geïnspireerd door de “map algebra” aanpak die wordt gebruikt in sommige veld-gebaseerde modelleerraamwerken. Dit is een handige aanpak voor het implementeren van modellen omdat de syntax van modelexpressies vergelijkbaar is met die van wiskundige expres-

sies, waarmee modelontwikkelaars al bekend zijn. Het hoge abstractieniveau is het resultaat van het wegabstraheren van de verschillen tussen ruimtelijke rasters en scalars, en door impliciet over alle cellen te itereren in de modelleeroperaties. Als een eerste stap richting een modelleerraamwerk die het geïntegreerd modelleren van agenten en velden ondersteunt, wordt een conceptueel datamodel beschreven, waarmee de verschillen tussen discrete objecten en continue velden worden weggeabstraheerd. Veel verschillende soorten aan objecten gerelateerde informatie, zoals hun locatie in de gesimuleerde tijd en plaats, hun eigenschappen, en hun onderlinge relaties, kunnen ermee worden gerepresenteerd. Velden kunnen er ook mee worden gerepresenteerd, als een ruimtelijke eigenschap van een “onderzoeksgebied” object. Het conceptuele datamodel kan als een blauwdruk dienen voor een type voor toestandvariabelen in een nieuw modelleerraamwerk en is als zodanig iets waar modelontwikkelaars mee zullen werken als ze hun modellen ontwikkelen. Een toekomstig modelleerraamwerk kan operaties ondersteunen die, behalve over rastercellen te itereren, ook itereren over alle objecten. Daarmee zou de syntax van simulatiemodellen vergelijkbaar kunnen blijven met huidige veld-gebaseerde modellen, maar integratie van agent- en veld-gebaseerd modelleren mogelijk worden.

Naast het conceptuele datamodel, wordt een fysisch datamodel beschreven waarmee de toestandvariabelen, georganiseerd volgens het conceptuele datamodel, kunnen worden opgeslagen in een dataset. De eigenlijke informatie waaraan gerefereerd wordt door toestandvariabelen kan erg verschillend zijn, afhankelijk van allerlei aspecten. De inhoud van een collectie objecten kan bijvoorbeeld variëren door de tijd, wanneer objecten worden “geboren” of “dood” gaan, of constant blijven. Daarnaast kunnen objecten mobiel zijn of stationair, en kunnen ze geen of meerdere eigenschappen hebben, waarvan sommige variëren door de tijd of constant zijn, etc. Drie abstractielagen zijn gedefinieerd waarmee deze diversiteit aan informatie over objecten kan worden georganiseerd in een dataset. Met de laagste abstractielaag kunnen zes algemene soorten arrays met temporele informatie worden opgeslagen, en met de hoogste abstractielaag kunnen de eigenlijke ruimtelijk-temporele objecten waaraan de toestandvariabelen refereren worden opgeslagen. Het fysische datamodel is geïmplementeerd als een bibliotheek met C++ en Python APIs bovenop de bovenste abstractielaag, en kan worden gebruikt om te lezen uit of te schrijven naar een HDF5 dataset.

SCHAALBARE BEREKENINGEN

In de loop van de tijd worden simulatiemodellen vaak groter, doordat de grootte van de toestandvariabelen toeneemt of het aantal modelleeroperaties, of beide. Dit gebeurt bijvoorbeeld wanneer een nieuwe satelliet in gebruik is genomen, die meer gedetailleerde informatie levert over de staat van de omgeving. Om groter wordende modellen nog steeds te kunnen executeren, moeten deze modellen de beschikbare hardware efficiënt kunnen gebruiken. Anders kan het zijn dat het executeren ervan te lang duurt of dat ze helemaal niet meer kunnen worden geëxecuteerd.

Goed schalende simulatiemodellen zijn in staat extra hardware efficiënt te gebruiken. Om een model dat met behulp van een modelleerraamwerk is ontwikkeld goed te laten schalen, moet het modelleerraamwerk goed schalen. Dit betekent dat de datastructuren en algoritmes ondersteuning bieden voor het ontwikkelen van schaalbare modellen, ongeacht de combinatie van toestandvariabelen en modelleeroperaties.

Een aanpak om schaalbare software te ontwikkelen die met name veelbelovend is in de context van modelleerraamwerken, is de “asynchronous many-tasks” (AMT) aanpak. Hiermee kan de softwareontwikkelaar taken definiëren die elk een deel van het werk uitvoeren, en de afhankelijkheden tussen deze taken. De AMT runtime is ervoor verantwoordelijk dat deze taken zo snel mogelijk worden uitgevoerd, gebruik makend van alle beschikbare hardware, en rekening houdend met hun onderlinge afhankelijkheden. Enkele potentiële voordelen van AMT in de context van een modelleerraamwerk is dat er minder synchronisatiepunten neigen te zijn, die de schaalbaarheid beperken, en dat onafhankelijke taken van meerdere modelleeroperaties tegelijkertijd ingedeeld kunnen worden om te worden uitgevoerd, waarmee een deel van de onbalans in de werklast die mogelijk aanwezig is verborgen wordt.

In het tweede deel van deze thesis wordt gefocust op het probleem van hoe de AMT aanpak kan worden gebruikt in het ontwerp en de implementatie van een modelleerraamwerk, gegeven de eis dat modellen die ermee worden ontwikkeld goed presteren en schalen.

Om continue velden te representeren in modellen is een gepartitioneerde array ontworpen en geïmplementeerd. Om processen te representeren is een set van lokale, focale, zonale, globale, en stromingsnetwerkoperaties ontworpen en geïmplementeerd, die gepartiti-

oneerde arrays als argumenten accepteren en nieuwe gepartitioneerde arrays teruggeven. Een nuttige eigenschap van het gebruik van AMT is dat modelleeroperaties sneller worden uitgevoerd dan het werk dat asynchroon door hen wordt geïnitieerd. Dit maakt het mogelijk dat onafhankelijk werk van opeenvolgende modelleeroperaties gelijktijdig kan worden uitgevoerd.

De stromingsnetwerkoperaties zijn geïmplementeerd in termen van taken die de verantwoordelijkheid hebben om een resultaat te berekenen voor een enkele partitie. Deze taken hangen alleen af van taken die directe buurpartities afhandelen, en informatie kan hiertussen worden uitgewisseld. Taken die partities afhandelen met daarin bovenstroomse delen van grootschalige stromen zijn eerder klaar dan taken die partities afhandelen met daarin benedenstroomse delen van grootschalige stromen, en de bijbehorende resultaatpartities kunnen al deelnemen in eventuele volgende modelleeroperaties.

Resultaten van experimenten laten zien dat de prestaties en schaalbaarheid van modellen die gemaakt zijn met het nieuwe raamwerk goed zijn. Extra hardware kan efficiënt worden gebruikt. Daarnaast liet een experiment waarin twee stromingsnetwerkoperaties werden uitgevoerd zien dat onbalans in de werklast van de eerste aanroep verborgen werd door werk dat was geïnitieerd door de tweede operatie.

De gepartitioneerde array datastructuur en de modelleeroperaties zijn geïmplementeerd in C++ en maken gebruik van de HPX AMT bibliotheek en runtime. Modelontwikkelaars kunnen de Python taal gebruiken om hun modellen te definiëren. Deze modellen kunnen ongewijzigd worden uitgevoerd op laptops en computerclusters.

GEÏNTEGREERD AGENT- EN VELD-GEBASEERD MODELLEREN

De conceptuele en fysische datamodelen voor het representeren van allerlei soorten toestandvariabelen die worden gebruikt in agent- en veld-gebaseerd modelleren kunnen worden gebruikt als een basis voor het representeren van toestandvariabelen met een geïntegreerd modelleerraamwerk waarmee discrete objecten en continue velden kunnen worden gemanipuleerd. De AMT aanpak die gebruikt is in de implementatie van een prototype modelleerraamwerk met daarin een gepartitioneerde array datastructuur en een set van lokale, focale, zonale, globale, en stromingsnetwerkoperaties, blijkt een goede aanpak te zijn. Modelontwikkelaars kunnen er grote modellen mee ontwikkelen die in staat zijn om extra hardware efficiënt te gebruiken, om

sneller modelresultaten te verkrijgen, of om grotere datasets te kunnen doorrekenen.

Er resteert nog een groot aantal onderwerpen die nader onderzoek behoeven. Een belangrijk onderwerp is hoe een model-ontwikkel interface kan worden ontwikkeld die makkelijk bruikbaar is voor model-ontwikkelaars, die alle complexiteit verbergt die niet gerelateerd is aan het definiëren van staat en processen, en waarmee geïntegreerde agent- en veld-gebaseerde modellen kunnen worden ontwikkeld. Een ander onderwerp is hoe AMT kan worden gebruikt in een modelleerraamwerk voor schaalbaar agent-gebaseerd modelleren en geïntegreerd agent- en veld-gebaseerd modelleren.

ACKNOWLEDGEMENTS

Until a few years ago, it had never been my intention to embark on a PhD research project. For almost 20 years I had been a happy research software engineer, as it is now called, at the Department of Physical Geography in Utrecht. But then a vacancy was announced for a PhD project, which I could not resist, and I decided to apply for it.

The first people I want to thank here are the ones that gave me the opportunity to start with this adventure: Steven de Jong, Derek Karssenberg, Marc van Kreveld, Deb Panja, and Marian Rossen. I am also grateful for the support I received from both the Department of Physical Geography and the Department of Information and Computing Sciences, and I thank the ITS department of the University for supporting and partly funding the project.

The project has been a very pleasant experience, especially when I forget about the moments it was not. For a large part, this (the pleasantness!) was due to the wonderful team of supervisors, consisting of Derek Karssenberg, Marc van Kreveld, and Deb Panja. I learned a lot from our weekly discussions, but they were also great fun. Derek, Marc, and Deb are genuinely nice people to work with.

Periods of frustration are pretty much guaranteed during a PhD project, and I got my share of it. Some of it resulted from climbing the learning curve required to use a nice and novel software library, crafted by a group of smart people. Luckily, they are not only smart, but also very helpful. I especially want to thank John Biddiscombe, Hartmut Kaiser, and Mikael Simberg, whose answers to my questions have had a direct positive effect on my results.

A few persons have been especially influential in shaping my career. They are therefore at least indirectly responsible for me getting to this point. I am grateful for the interactions I have had over the years with Peter Burrough, Willem van Deursen, Victor Jetten, Tom de Jong, Derek Karssenberg (again ;-)), Edzer Pebesma, Oliver Schmitz, and Cees Wesseling.

When not in the office, people sometimes try to convince me that array-oriented design and asynchronous many-tasks are not the solution to many important problems in life. I have spontaneous—but weekly—meetings with the Chatham House fellowship in a local pub,

about which I cannot say anything of course (Chatham House Rule!). Thanks Gwenn, Henk, and Edwin for joining me during the weekly transition from workweek to weekend. Thanks also to the soccer teams of AZC 35+2 and 45+1, for the weekly shot of great fun.

Finally, I want to thank my friends and family, for expressing an interest in my research project the last few years. It is done! I will try to be less preoccupied from now on. This is especially relevant for Gwenn, Jesse, and Mink, who had to put up with a nerdy husband and father every day. Let's go cycle somewhere far away!



None of this—and quite a lot more—would have happened if it wasn't for us to exist at all. Therefore, I thank life itself. And the universe in which we live. And everything else as well. Most of it is so fascinating.

CURRICULUM VITAE

Kor was born on April 17th 1971 in Oosterwolde. In 1990 he moved to Utrecht to study Physical Geography. He specialized in Geographical Information Systems and enjoyed developing software for spatial applications. In 1996 he started working as a software developer for ARIS—a company specialized in Geographical Information Systems. After two years, Kor returned to the Department of Physical Geography, to work on projects related to the PCRaster environmental modelling software. In 2006 he moved, together with his family, for a year to sunny California, to work for Esri on a project to enhance the raster modelling capabilities of ArcGIS. Afterwards, Kor returned to work at the Department again, but stayed employed at Esri as well. In 2013 Kor quit his job at Esri, founded the Geoneri company, and became a part-time entrepreneur. In 2018 he got the opportunity to work full-time on the research project that ultimately resulted in this thesis. Currently Kor divides his professional time again between projects at the Department of Physical Geography and Geoneri.

JOURNAL PUBLICATIONS

- K. de Jong, D. Panja, D. Karssenbergh, and M. van Kreveld. "Scalability and composability of flow accumulation algorithms based on asynchronous many-tasks." In: *Computers & Geosciences* 162 (2022), p. 105083. doi: 10.1016/j.cageo.2022.105083.
- K. de Jong, D. Panja, M. van Kreveld, and D. Karssenbergh. "An environmental modelling framework based on asynchronous many-tasks: scalability and usability." In: *Environmental Modelling & Software* 139 (2021), p. 104998. doi: 10.1016/j.envsoft.2021.104998.
- K. de Jong and D. Karssenbergh. "A physical data model for spatio-temporal objects." In: *Environmental Modelling & Software* 122 (2019), p. 104553. doi: 10.1016/j.envsoft.2019.104553.
- E. H. Sutanudjaja, R. van Beek, N. Wanders, Y. Wada, J. H. C. Bosmans, N. Drost, R. J. van der Ent, I. E. M. de Graaf, J. M. Hoch, K. de Jong, D. Karssenbergh, P. L. López, S. Peßenteiner, O. Schmitz, M. W. Straatsma, E. Vannamettee, D. Wisser, and M. F. P. Bierkens. "PCR-

- GLOBWB 2: A 5 arcmin global hydrological and water resources model." In: *Geoscientific Model Development* 11.6 (2018), pp. 2429–2453. DOI: 10.5194/gmd-11-2429-2018.
- M. P. de Bakker, K. de Jong, O. Schmitz, and D. Karssenberg. "Design and demonstration of a data model to integrate agent-based and field-based modelling." In: *Environmental Modelling & Software* 89 (2017), pp. 172–189. DOI: 10.1016/j.envsoft.2016.11.016.
- O. Schmitz, D. Karssenberg, K. de Jong, J.-L. de Kok, and S. M. de Jong. "Map algebra and model algebra for integrated model building." In: *Environmental Modelling & Software* 48 (2013), pp. 113–128. DOI: 10.1016/j.envsoft.2013.06.009.
- D. Karssenberg, O. Schmitz, P. Salamon, K. de Jong, and M. F. Bierkens. "A software framework for construction of process-based stochastic spatio-temporal models and data assimilation." In: *Environmental Modelling & Software* 25.4 (2010), pp. 489–502. DOI: 10.1016/j.envsoft.2009.10.004.
- D. Karssenberg, K. de Jong, and J. van der Kwast. "Modelling landscape dynamics with Python." In: *International Journal of Geographical Information Science* 21.5 (2007), pp. 483–495. DOI: 10.1080/13658810601063936.
- E. J. Pebesma, K. de Jong, and D. Briggs. "Interactive visualization of uncertain spatial and spatio-temporal data under different scenarios: an air quality example." In: *International Journal of Geographical Information Science* 21.5 (2007), pp. 515–527. DOI: 10.1080/13658810601064009.
- I. Thonon, K. de Jong, M. van der Perk, and H. Middelkoop. "Modelling floodplain sedimentation using particle tracking." In: *Hydrological Processes* 21.11 (2007), pp. 1402–1412. DOI: 10.1002/hyp.6296.
- D. Karssenberg and K. de Jong. "Dynamic environmental modelling in GIS: 1. Modelling in three spatial dimensions." In: *International Journal of Geographical Information Science* 19.5 (2005), pp. 559–579. DOI: 10.1080/13658810500032362.
- D. Karssenberg and K. de Jong. "Dynamic environmental modelling in GIS: 2. Modelling error propagation." In: *International Journal of Geographical Information Science* 19.6 (2005), pp. 623–637. DOI: 10.1080/13658810500104799.

- P. H. Verburg, T. C. de Nijs, J. R. van Eck, H. Visser, and K. de Jong. "A method to analyse neighbourhood characteristics of land use patterns." In: *Computers, Environment and Urban Systems* 28.6 (2004), pp. 667–690. doi: 10.1016/j.compenvurbsys.2003.07.001.
- D. Karssenbergh, P. A. Burrough, R. Sluiter, and K. de Jong. "The PCRaster software and course materials for teaching numerical modelling in the environmental sciences." In: *Transactions in GIS* 5.2 (2001), pp. 99–110. doi: 10.1111/1467-9671.00070.

BOOK CHAPTERS

- D. Ettema, K. de Jong, H. Timmermans, and A. Bakema. "Puma: multi-agent modelling of urban systems." In: *Modelling Land-Use Change: Progress and Applications*. Ed. by E. Koomen, J. Stillwell, A. Bakema, and H. J. Scholten. 2007, pp. 237–258. doi: 10.1007/978-1-4020-5648-2_14.
- D. Karssenbergh and K. de Jong. "Towards improved solution schemes for Monte Carlo simulation in environmental modeling languages." In: *Geo-Information and Computational Geometry*. Ed. by P. J. van Oosterom and M. J. van Kreveld. Nederlandse Commissie voor Geodesie, 2006, pp. 43–51.

SOFTWARE

- K. de Jong and O. Schmitz. *computationalgeography/lue: LUE-0.3.0: Scientific database and environmental modelling framework*. Version 0.3.0. Sept. 2021. doi: 10.5281/zenodo.5535686.
- PCRaster Research and Development Team. *PCRaster environmental modelling framework*. 2017. URL: <https://github.com/pcraster/pcraster> (visited on 01/17/2022).
- K. de Jong. *Fern software components for scalable geocomputing*. 2014. URL: <https://github.com/geoneric/fern> (visited on 01/17/2022).
- Esri. *ArcGIS Spatial Analyst*. 2013. URL: <https://www.esri.com> (visited on 01/17/2022).
- PCRaster Research and Development Team. *Aguila exploratory data analysis software*. 2006. URL: <https://github.com/pcraster/pcraster> (visited on 01/17/2022).

Utrecht University
Faculty of Geosciences
Department of Physical Geography