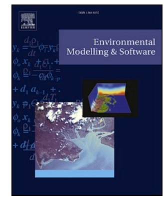
Environmental Modelling and Software 139 (2021) 104998

Contents lists available at ScienceDirect

Environmental Modelling and Software

journal homepage: http://www.elsevier.com/locate/envsoft



An environmental modelling framework based on asynchronous many-tasks: Scalability and usability

Kor de Jong [a,b,*], Debabrata Panja [b], Marc van Kreveld [b], Derek Karssenberg [a]

[a] Department of Physical Geography, Faculty of Geosciences, Utrecht University, Princetonlaan 8A, 3584, CB, Utrecht, the Netherlands
[b] Department of Information and Computing Sciences, Faculty of Science, Utrecht University, Princetonplein 5, 3584, CC, Utrecht, the Netherlands

ARTICLE INFO

Keywords:
Environmental modelling framework
Map algebra
High-performance computing
Asynchronous many-tasks
HPX
LUE

ABSTRACT

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.

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[1] containing pre-built model building blocks, like Google Earth Engine for earth science data and analysis (Gorelick et al. (2017)), MATLAB (Holzbecher (2012)), the NetLogo agent-based modelling framework (Wilensky (1999)), and the PCRaster field-based modelling framework (Karssenberg et al. (2010)). 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 (Fig. 1 and Karssenberg (2002)).

Some of the model development interfaces are inspired by map algebra (Tomlin (1990)), 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 (Karssenberg et al. (2010)). 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

* Corresponding author. Department of Physical Geography, Faculty of Geosciences, Utrecht University, Princetonlaan 8A, 3584, CB, Utrecht, the Netherlands.
E-mail addresses: k.dejong1@uu.nl (K. de Jong), d.panja@uu.nl (D. Panja), m.j.vankreveld@uu.nl (M. van Kreveld), d.karssenberg@uu.nl (D. Karssenberg).

[1] We use the term framework loosely, to mean software containing at least data types and algorithms, used for the development of individual models. This includes the case of a software library implementing these, but excludes integration frameworks used for coupling models.

https://doi.org/10.1016/j.envsoft.2021.104998
Accepted 13 February 2021
Available online 25 February 2021
1364-8152/© 2021 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

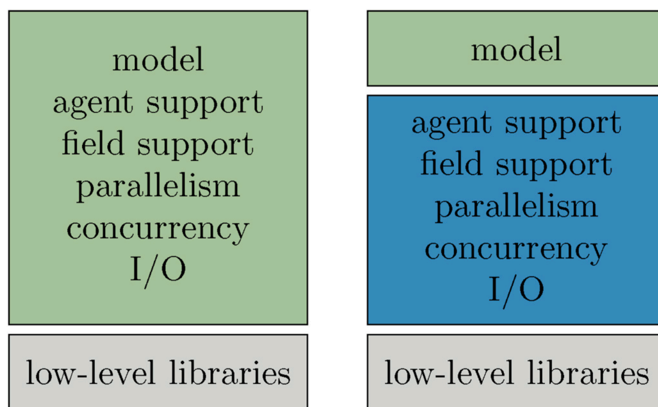


Fig. 1. Two stacks of model and support code. A model developer writing a model from scratch has to write more code (left) than a developer using a modelling framework (right). The green boxes represent the amount of code the model developer has to write. The blue box represents the framework code.

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 (Dagum and Menon (1998)) 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, Guan and Clarke (2010)), the Parallel Raster-based Geocomputation Operators (PaRGO Qin et al. (2014)), and the Parallel Cartographic Modelling Language (PCML, Shook et al. (2016)). 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 (Chamberlain et al. (2007)), X10 (Charles et al. (2005)), HPX (Kaiser et al. (2020a)) and Charm++ (Kale and Krishnan (1993)), 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. (2018). 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 paper is organized as follows: in Section 2 we describe the approach of developing environmental models using map algebra in more detail; in Section 3 we provide more information about AMT and the HPX implementation thereof; in Section 4 we describe how we used AMT to implement a map algebra development interface on top of modelling algorithms; in Section 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 paper with a discussion of the results in Section 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 (Section 7).

2. Model development using map algebra

Originally, the map algebra language was designed for creating cartographic models, where the models were collections of maps (Tomlin (1990)). 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 (van Deursen et al. (2019); Karssenberg et al. (2010)). 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 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). Outputs from the model are shown in (Fig. 2).

Map algebra operations are often classified according to the kind of

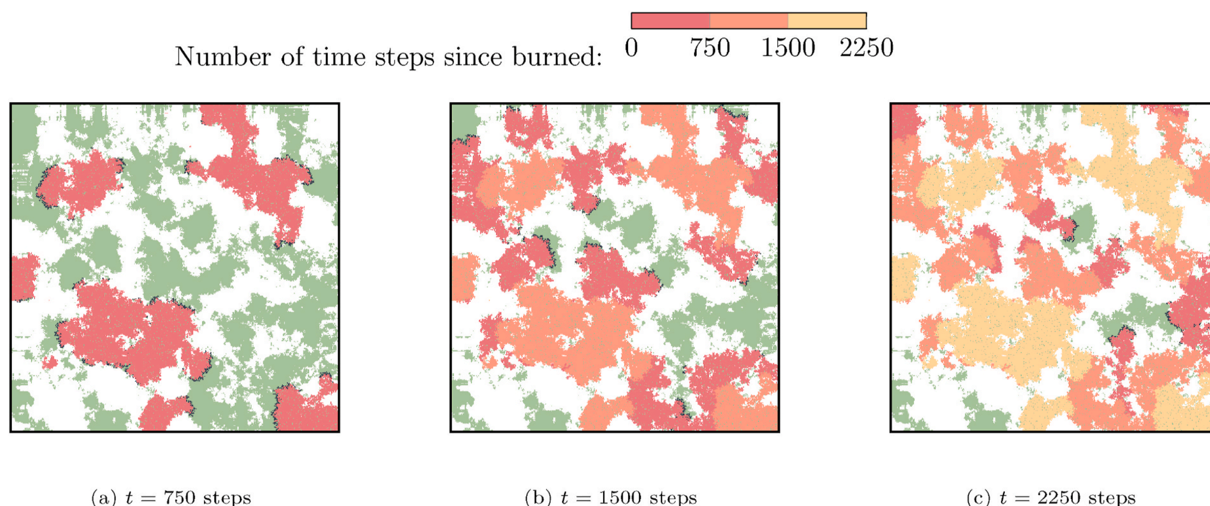


Fig. 2. Output maps of the example model simulating wildfire for three time steps. Black cells represent cells that are burning (fire front). Green cells represent the area with the highest susceptibility for being burned. The other colours represent the age of burned cells in number of time steps, with red cells being burned most recently. The area shown is 500 × 500 cells.

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

Listing 1. Pseudocode of model simulating wildfire.

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 (Burrough et al. (2015); Karssenberg et al. (2010); Tomlin (1990)): local operations (Fig. 3a), focal operations (Fig. 3b), and zonal operations (Fig. 3c). Operations not considered in this work include global operations, which can be seen as a subset of zonal operations, and network operations, operating on a flow direction network.

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,

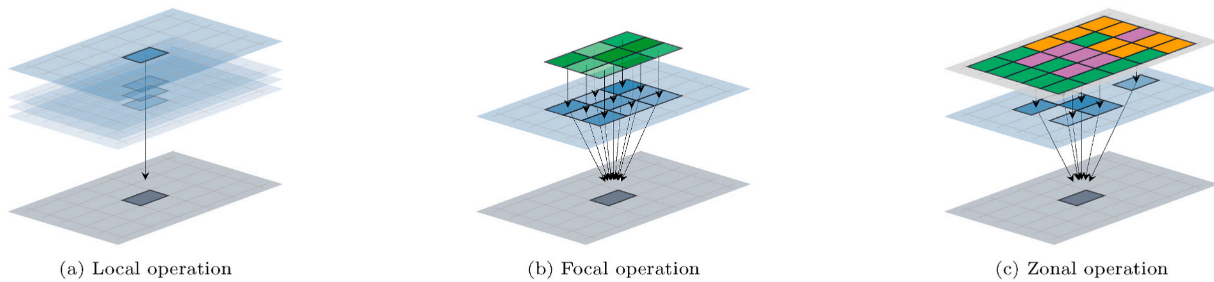


Fig. 3. a: Each output cell is a function of input cells at the corresponding location in one or more input rasters. Example: $ph = -\log(hydronium)$ b: Each output cell is a function of input cells within a neighbourhood of input cells at and around the corresponding location in an input raster. In 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 runtime system will execute concurrent tasks in parallel.

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 (Heller et al. (2013), Heller et al. (2017), Heller et al. (2019), and Khatami et al. (2016)).

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 (Grubel et al. (2015)). 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 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 (Fig. 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 (Fig. 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 (Fig. 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 (Fig. 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,

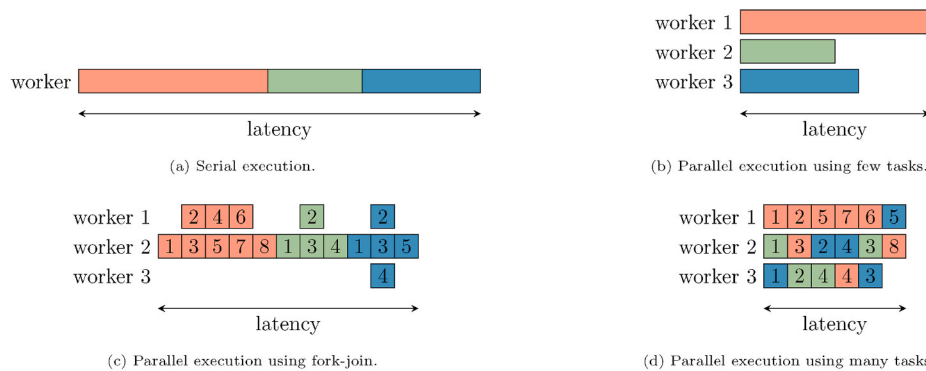


Fig. 4. a: Serial execution of three operations by one worker. Each colour represents an operation. Latency is the duration of executing all the work. b: Parallel execution of three operations by three workers. c: Serial execution of three operations, but parallel execution by three workers of the concurrent tasks within each operation (Fig. 5). d: Parallel execution by three workers of concurrent tasks of all operations.

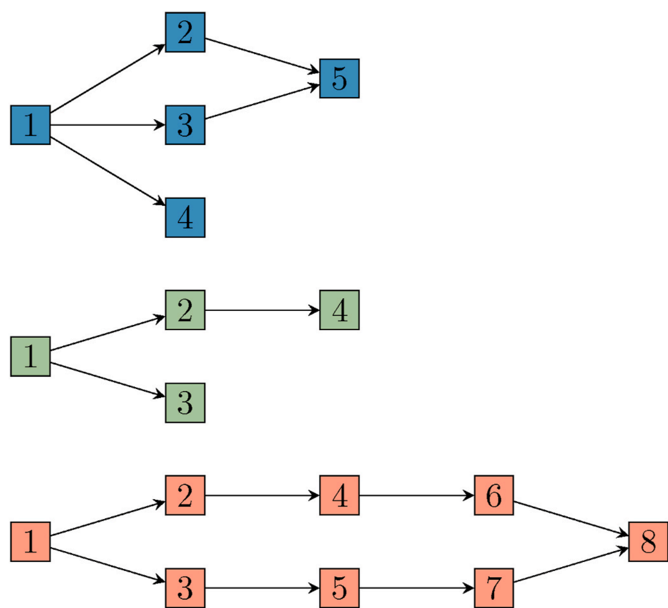


Fig. 5. Each operation has concurrent tasks that can be executed in parallel. Each colour represents an operation and each numbered box represents a task. The length of each task represents its latency which, for simplicity, is assumed to be constant.

OpenMP (Dagum and Menon (1998)) 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 (Fig. 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 Fig. 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 (MPI-Forum (2015)) for the distribution and OpenMP for the parallelization of work.

4. Method

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 Fig. 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

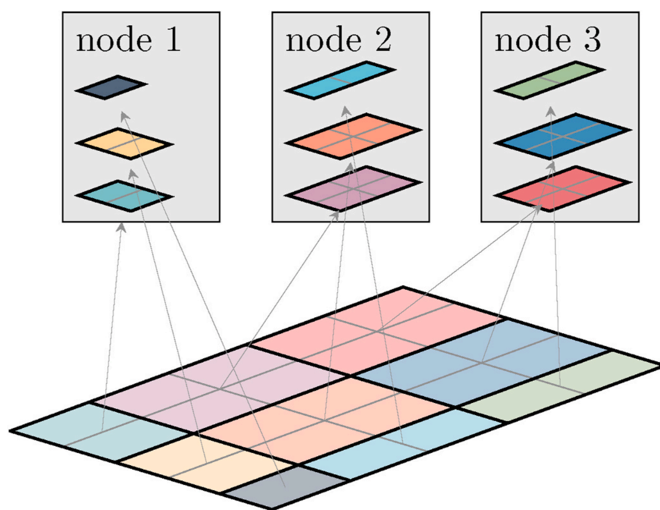


Fig. 6. A partitioned array instance is stored in a single process and contains an array of partition client instances. Each of these refers to a component server instance actually containing the array elements, and which is possibly located in a different process. Each partition's elements are ready to be used, are currently being calculated, or will be calculated in the future.

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 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 A. 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.

Table 1
Hardware and software platform of nodes used in experiments. All nodes are interconnected with InfiniBand.

CPUs	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	8192 KiB/3 cores
RAM	256 GiB (32 GiB/NUMA node)
OS	CentOS 7
GNU GCC	version 10.2.0
HPX	version 1.5.0 (Kaiser et al. (2020b))

4.2. Scalability and performance

To characterize the scalability of our modelling framework, we developed an example model, simulating wildfire (Listing 1). This model is based on concepts from existing fire models (e.g. Clarke et al. (1994); Li et al. (2017); Freire and DaCamara (2019); Trucchia et al. (2020)) 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 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 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 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 (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.

Table 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.

	individual operations		model	
Worker	strong	weak	strong	weak
Core	$10,000 \times 10,000$	$4,000 \times 4,000$	$5,000 \times 5,000$	$2,000 \times 2,000$
NUMA node	$30,000 \times 30,000$	$10,000 \times 10,000$	$15,000 \times 15,000$	$5,000 \times 5,000$
cluster node	$96,000 \times 96,000$	$30,000 \times 30,000$	$48,000 \times 48,000$	$15,000 \times 15,000$

$$\text{strong scaling efficiency} = \frac{T_{S,1}}{P \times T_{S,P}} \times 100\% \tag{1}$$

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 (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.

$$\text{weak scaling efficiency} = \frac{T_{W,1}}{T_{W,P}} \times 100\% \tag{2}$$

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 focussed on the scalability of the computations. Time spent on I/O was not taken into account when measuring latencies.

In Table 2 the size of the arrays used in the scaling experiments are shown. The sizes where 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 $30\times$ 30m 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 3 h to finish.

In Appendix B 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 1.

Because scalability is the main focus in this work, 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 (Karssenberg et al. (2010)).

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 (ISO/IEC 25010:2011 (2011)). 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.

5. Results

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 (Fig. 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 3). The experiments also show that the variability in latencies is relatively small at optimal partition sizes (Fig. 7). We therefore did not perform the strong and weak scaling experiments multiple times.

In general, the strong and weak scaling experiments show good scalability (Fig. 8 and Table 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). 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 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 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 PCRaster.[2] In an experiment using a single CPU core, with rasters of 500×500 cells and 5000 time steps, PCRaster took 5 min and the new framework 5:45 min. These experiments used a single CPU core, and performed I/O, to different file formats.

The latencies shown in Table 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 2), the weak scaling latencies increase when going from CPU cores to NUMA

[2] The PCRaster model is available in the source code repository associated with this paper (Section 7).

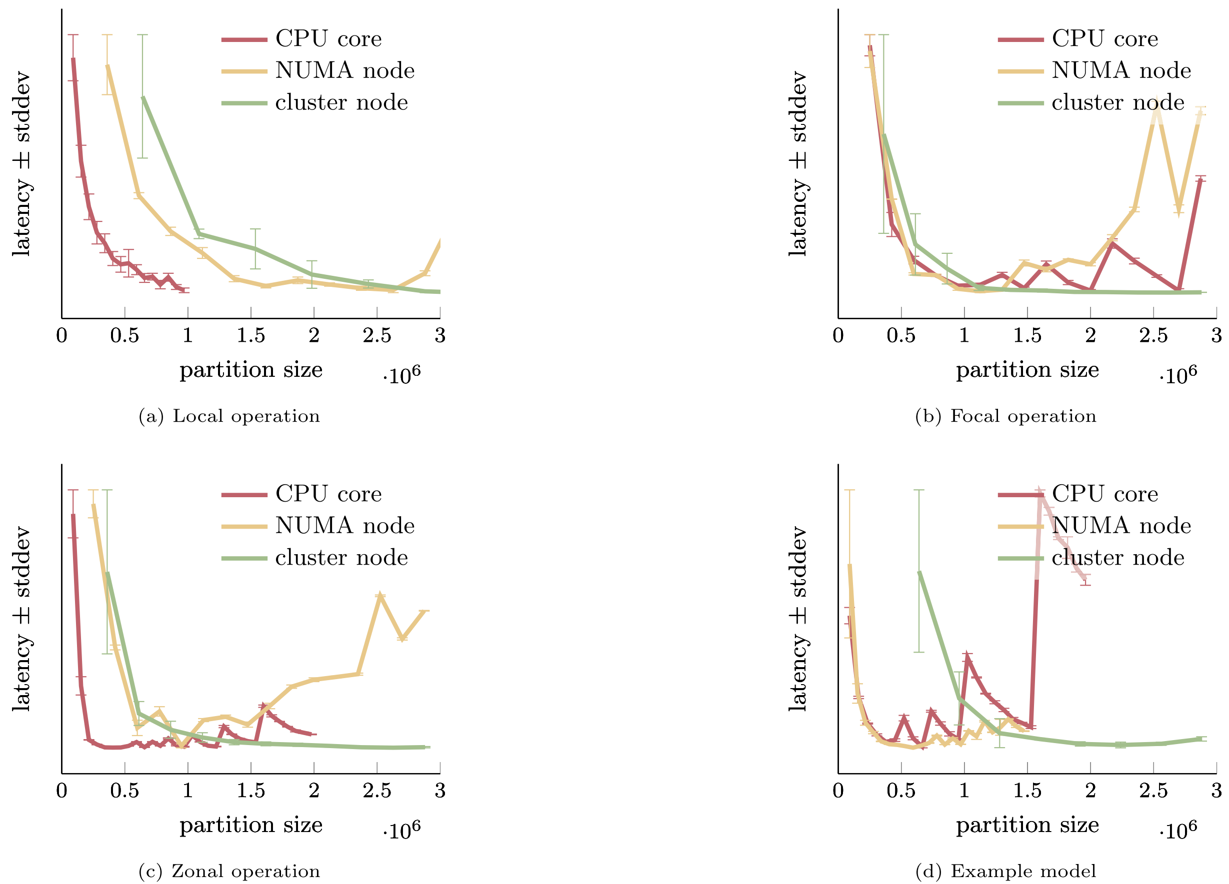


Fig. 7. Latency of experiments for different kinds of workers and partition sizes. Partition sizes shown are the number of cells in square partitions.

Table 3
Sizes of partitions used in the scaling experiments. These sizes correspond with the smallest sizes in Fig. 7, where the latencies are small.

Worker	local	focal	zonal	Model
Core	$1,000 \times 1,000$	$1,000 \times 1,000$	600×600	650×650
NUMA node	$1,300 \times 1,300$	$1,000 \times 1,000$	975×975	750×750
cluster node	$1,900 \times 1,900$	$1,200 \times 1,200$	$1,600 \times 1,600$	$1,500 \times 1,500$

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.

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 A.

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.

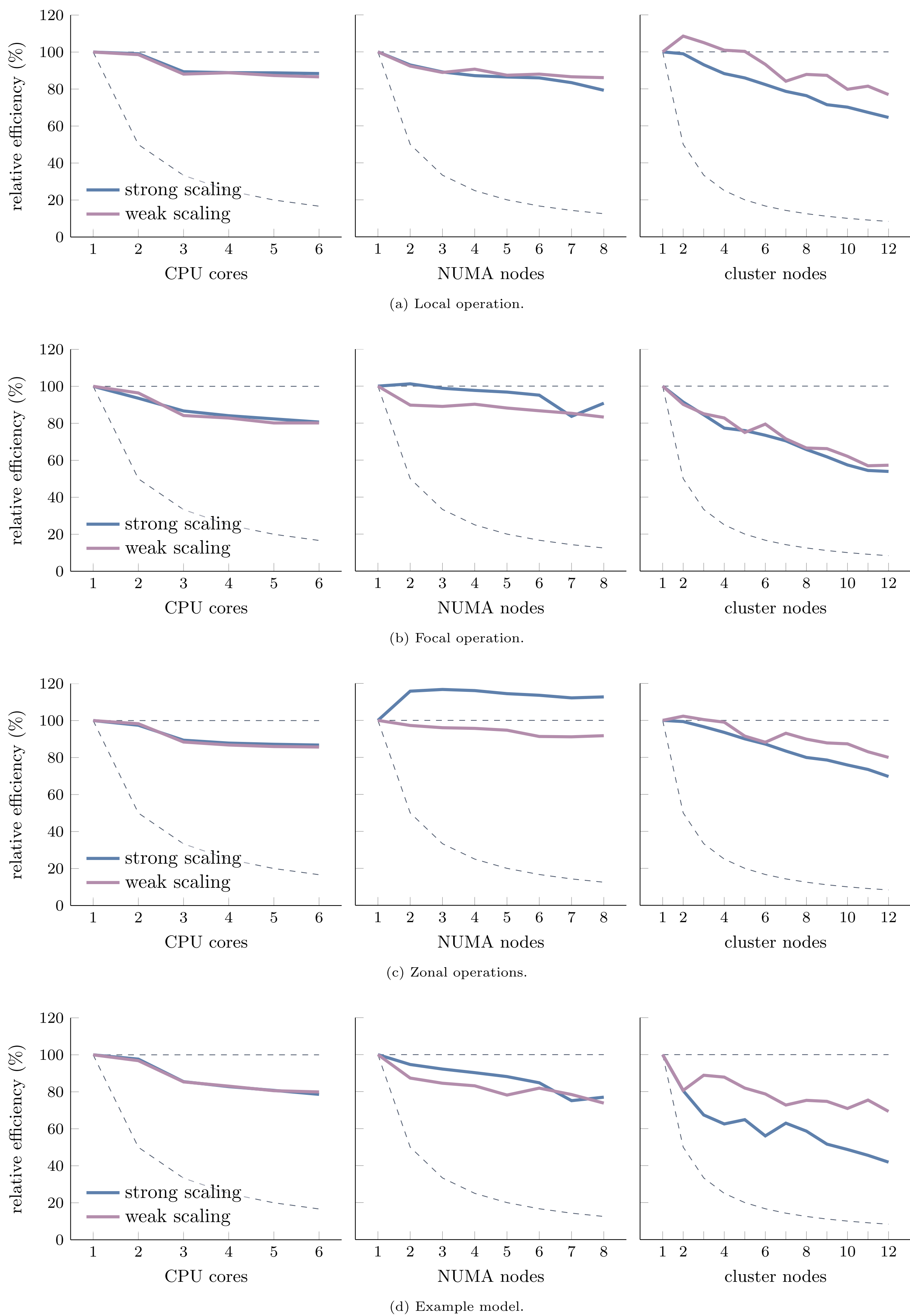


Fig. 8. Scaling efficiencies of experiments, over different kinds of workers. For reference, efficiencies for linear scaling (upper dashed line) and serial scaling (lower dashed line) are also shown.

Table 4
Scaling efficiencies of strong and weak scaling experiments when using the maximum number of workers.

	strong scaling				weak scaling			
Workers	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 5
Throughputs in MLUPS (million lattice updates per second) of weak scaling experiments when using the maximum number of workers.

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 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 2).

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

6. Discussion

6.1. Scalability and performance

In this paper 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 work 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. (2019) 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 focussed 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 (MPI-Forum (2015)) and higher level APIs using it, like NetCDF (UNIDATA (2008–2018)), HDF5 (The HDF Group (1997–2019)), or the LUE physical data model (de Jong and Karssenberg (2019)). The latter API is part of the same software library as the framework described in this paper. 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 (Shook et al. (2016)) 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.

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 B. 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 (Huck et al. (2015)) 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, Section 7). 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, 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 3072 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 2000 $\times$ 2000 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 inter-dependent 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.

6.3. Future work

Given the promising results of this work, we will continue adding more functionality and improving the existing functionality in our future work. 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 work. Examples of such operations are those that operate on a hydrologic flow direction network, and operations that operate on a friction-distance path surface.

7. Software availability

The scalable modelling framework is implemented as part of a software package called LUE,[3] which is hosted on GitHub at https://github.com/computationalgeography/lue. The framework is implemented by Kor de Jong (corresponding author) in C++ and the source code is freely available under the MIT open source license.

A document called README.md is included in the root of the source code repository detailing the instructions for building the software. LUE is portable software and has been successfully built on various platforms (operating systems: Linux, macOS; compilers: Clang, GCC; architecture: x86-64).

A project containing the version of LUE used in this work, and containing additional information about the commands used for the described experiments can also be found on GitHub, at https://github.com/computationalgeography/paper_2020_scalable_algorithms.

A release of LUE Python packages for various platforms is planned for 2021.

CRediT authorship contribution statement

Kor de Jong: Conceptualization, Writing – original draft, worked on concepts, designed and implemented the framework, wrote the manuscript. **Debabrata Panja:** Writing – original draft, wrote the manuscript. **Marc van Kreveld:** Writing – original draft, wrote the manuscript. **Derek Karssenberg:** Writing – original draft, wrote the manuscript.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Funding: This work was supported by the Research IT innovation programme (Utrecht University, The Netherlands).

[3] 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. We pronounce LUE as the French pronounce Louis (LU-EE).

Appendix A. 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

```
// 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
}

// 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
}
```

Appendix B. Experiments

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 paper (Section 7).

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.

```
# 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 2. Pseudocode of local 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 3. Pseudocode of focal 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)
```

Listing 4. Pseudocode of zonal operation experiment.

References

Burrough, P., McDonnell, R.A., Lloyd, C.D., 2015. Principles of Geographical Information Systems, third ed. Oxford University Press.

Chamberlain, B., Callahan, D., Zima, H., 2007. Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. 21 (3), 291–312. https://doi.org/10.1177/1094342007078442. URL.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V., 2005. 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. OOPSLA '05. Association for Computing Machinery, New York, NY, USA, pp. 519–538. https://doi.org/10.1145/1094811.1094852. URL.

Clarke, K., Brass, J., Riggan, P., 1994. A cellular automaton model of wildfire propagation and extinction. Photogramm. Eng. Rem. Sens. 60 (11), 1355–1367.

Dagum, L., Menon, R., 1998. OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE 5 (1), 46–55.

de Jong, K., Karssenberg, D., 2019. A physical data model for spatio-temporal objects. Environ. Model. Software 122, 104553. https://doi.org/10.1016/j.envsoft.2019.104553. URL.

Freire, J.G., DaCamara, C.C., 2019. Using cellular automata to simulate wildfire propagation and to assist in fire management. Nat. Hazards Earth Syst. Sci. 19 (1), 169–179. https://doi.org/10.5194/nhess-19-169-2019. URL.

Gorelick, N., Hancher, M., Dixon, M., Ilyushchenko, S., Thau, D., Moore, R., 2017. Google earth engine: planetary-scale geospatial analysis for everyone. Rem. Sens. Environ. 202, 18–27.

Grubel, P., Kaiser, H., Cook, J., Serio, A., 2015. The performance implication of task size for applications on the HPX runtime system. In: 2015 IEEE International Conference on Cluster Computing. IEEE, pp. 682–689. https://doi.org/10.1109/CLUSTER.2015.119. URL.

Guan, Q., Clarke, K.C., 2010. A general-purpose parallel raster processing programming library test application using a geographic cellular automata model. Int. J. Geogr. Inf. Sci. 24 (5), 695–722. https://doi.org/10.1080/13658810902984228. URL.

Heller, T., Adelstein Lelbach, B., Huck, K., Biddiscombe, J., Grubel, P., Koniges, A., Kretz, M., Marcello, D., Pfander, D., Serio, A., Frank, J., Clayton, G., Pflüger, D., Eder, D., Kaiser, H., 2019. Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger of two stars. Int. J. High Perform. Comput. Appl. 33 (4), 699–715. https://doi.org/10.1177/1094342018819744. URL.

Heller, T., Diehl, P., Byerly, Z., Biddiscombe, J., Kaiser, H., 2017. HPX – an open source C++ standard library for parallelism and concurrency. In: OpenSuCo 2017, p. 5. Denver, Colorado USA, November 2017.

Heller, T., Kaiser, H., Iglberger, K., 2013. Application of the ParalleX execution model to stencil-based problems. Comput. Sci. Res. Dev. 28, 253–261. https://doi.org/10.1007/s00450-012-0217-1. URL.

Holzbecher, E., 2012. Environmental Modeling, second ed. Springer-Verlag Berlin-Heidelberg.

Huck, K., Porterfield, A., Chaimov, N., Kaiser, H., Malony, A., Sterling, T., Fowler, R., 2015. An autonomic performance environment for exascale. Supercomputing Frontiers and Innovations 2 (3). https://doi.org/10.14529/jsfi150305. URL.

ISO/IEC 25010:2011, 2011. Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models. Standard. International Organization for Standardization, Geneva, CH.

Kaiser, H., Diehl, P., Lemoine, A.S., Lelbach, B.A., Amini, P., Berge, A., Biddiscombe, J., Brandt, S.R., Gupta, N., Heller, T., Huck, K., Khatami, Z., Kheirkhahan, A., Reverdell, A., Shirzad, S., Simberg, M., Wagle, B., Wei, W., Zhang, T., 2020a. HPX - the C++ standard library for parallelism and concurrency. Journal of Open Source Software 5 (53), 2352. https://doi.org/10.21105/joss.02352. URL.

Kaiser, H., Lelbach, B.A., Simberg, M., Heller, T., Bergé, A., Biddiscombe, J., Bikineev, A., Mercer, G., Schäfer, A., Huck, K., Lemoine, A.S., Kwon, T., Habraken, J., Anderson, M., Copik, M., Brandt, S.R., Stumpf, M., Bourgeois, D., Blank, D., rstobaugh, Jakobovits, S., Amatya, V., Viklund, L., Gupta, N., Diehl, P., Khatami, Z., Bacharwar, D., Tapasweni Pathak, S.Y., 2020b. STEllAR-GROUP/Hpx: HPX V1.5.0: the C++ Standards Library for Parallelism and Concurrency. R., A.. https://doi.org/10.5281/zenodo.4011590, 9, URL.

Kale, L.V., Krishnan, S., 1993. CHARM++: a portable concurrent object oriented system based on C++. In: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA '93. Association for Computing Machinery, New York, NY, USA, pp. 91–108. https://doi.org/10.1145/165854.165874. URL.

Karssenberg, D., 2002. Building Dynamic Spatial Environmental Models. Utrecht University, The Netherlands. Ph.D. thesis.

Karssenberg, D., Schmitz, O., Salamon, P., de Jong, K., Bierkens, M.F., 2010. A software framework for construction of process-based stochastic spatio-temporal models and data assimilation. Environ. Model. Software 25 (4), 489–502.

Khatami, Z., Kaiser, H., Grubel, P., Serio, A., Ramanujam, J., 2016. A massively parallel distributed N-body application implemented with HPX. In: 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems. IEEE, pp. 57–64.

Li, Z., Wang, F., Zheng, X., Jiang, W., Meng, Q., Liu, B., 2017. 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. EM-GIS'17. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3152465.3152470. URL.

MPI-Forum 6, 2015. MPI: a message-passing interface standard. In: High-Performance Computing Center Stuttgart. University of Stuttgart. URL, Version 3.1. https://www.mpi-forum.org.

Qin, C.-Z., Zhan, L.-J., Zhu, A.-X., Zhou, C.-H., 2014. A strategy for raster-based geocomputation under different parallel computing platforms. Int. J. Geogr. Inf. Sci. 28 (11), 2127–2144. https://doi.org/10.1080/13658816.2014.911300. URL.

Shook, E., Hodgson, M.E., Wang, S., Behzad, B., Soltani, K., Hiscox, A., Ajayakumar, J., 2016. Parallel cartographic modeling: a methodology for parallelizing spatial data processing. Int. J. Geogr. Inf. Sci. 30 (12), 2355–2376. https://doi.org/10.1080/13658816.2016.1172714. URL.

The HDF Group, 1997. Hierarchical Data Format, Version 5, 2019. URL http://www.hdfgroup.org/HDF5/. (Accessed 12 July 2019).

Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., Gschwandtner, P., Lemarinier, P., Markidis, S., Jordan, H., Fahringer, T., Katrinis, K., Laure, E., Nikolopoulos, D.S., 2018. A taxonomy of task-based parallel programming technologies for high-performance computing. J. Supercomput. 74, 1422–1434. https://doi.org/10.1007/s11227-018-2238-4. URL.

Tomlin, D., 1990. Geographic Information Systems and Cartographic Modeling, first ed. Prentice Hall.

Trucchia, A., D'Andrea, M., Baghino, F., Fiorucci, P., Ferraris, L., Negro, D., Gollini, A., Severino, M., 2020. PROPAGATOR: an operational cellular-automata based wildfire simulator. Fire 3 (3). https://doi.org/10.3390/fire3030026. URL.

UNIDATA, 2008-2018. Network common data form. URL. https://www.unidata.ucar.edu/software/netcdf/docs/index.html. (Accessed 12 July 2019). version 4.

van Deursen, W., Wesseling, C., Karssenberg, D., de Jong, K., Schmitz, O., 2019. The PCRaster environmental modelling framework. URL. https://pcraster.computationalgeography.org.

Wilensky, U., 1999. NetLogo. Center for Connected Learning and Computer-Based Modeling. Northwestern University, Evanston, IL, USA available at: http://ccl.northwestern.edu/netlogo/. (Accessed 12 July 2019).