Computers & Geosciences 162 (2022) 105083

Contents lists available at ScienceDirect

Computers and Geosciences

journal homepage: www.elsevier.com/locate/cageo

Research paper

Scalability and composability of flow accumulation algorithms based on asynchronous many-tasks

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

[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:
Modelling framework
Flow accumulation
High-performance computing
Asynchronous many-tasks
HPX
LUE

ABSTRACT

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.

1. Introduction

The changing state of the biophysical environment through time and space can be simulated using computer models. Modelling frameworks[1] 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 (Burrough et al., 2015).

The current paper 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 (O'Callaghan and Mark, 1984). 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 (Burek et al., 2013), used for the European Flood Awareness System (EFAS), and the PCR-GLOBWB global water balance model (Sutanudjaja et al., 2017).

Flow accumulation algorithms that can be found in the literature solve the problem of transporting *all* material in downstream direction (Ortega and Rueda, 2010; Sten et al., 2016; Barnes, 2017; Cordonnier et al., 2019; Zhou et al., 2019; Kotyra et al., 2021), but sometimes flow

* Corresponding author at: 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), d.karssenberg@uu.nl (D. Karssenberg), m.j.vankreveld@uu.nl (M. van Kreveld).

[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.cageo.2022.105083
Received 15 October 2021; Received in revised form 8 February 2022; Accepted 25 February 2022
Available online 18 March 2022

Table 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.

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

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 1, (Karssenberg, 2006)). 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 Hortonian overland flow (Hendriks, 2010) 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. (2019) 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. (2021) it has been concluded that a parallel version of the algorithm by Zhou et al. (2019) performs best compared to other parallel algorithms they tested; one limitation of their algorithm is that is not capable of using multiple nodes in a compute cluster. Barnes (2017) 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. (2019), 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 (2017) 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 (Kaiser et al., 2020). 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[2] modelling framework (de Jong et al., 2021), 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 paper is as follows. We start with an introduction of AMT, HPX and the LUE modelling framework (Section 2). We then describe our flow accumulation algorithms (Section 3), and the experiments we performed (Section 4). The results of the experiments can be found in Section 5. We finish the paper with a discussion of the results and our conclusions (Section 6).

[2] 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.

2. AMT, HPX, And the LUE modelling framework

We start with a brief introduction of some major aspects of AMT, the HPX implementation thereof (Kaiser et al., 2020), and the use of AMT and HPX in the LUE environmental modelling framework (de Jong et al., 2021).

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 runtime. It is an open source software library written in portable C++ 11/14/17/20 code. With HPX, every system, ranging from laptops to compute 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 (Tomlin, 1990). 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 compute clusters. To the modeller, LUE models look similar to models created with other map algebra implementations, like ArcGIS (Esri, 2021), GDAL (GDAL/OGR contributors, 2021), GRASS (Neteler et al., 2012), and PCRaster (van Deursen et al., 2019). 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.

3. Flow accumulation

Our flow accumulation algorithms combine and extend the efficient algorithm of Zhou et al. (2019) and the distributed algorithm of Barnes (2017). 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.

3.1. Overview

Our algorithm works with partitioned arrays (Section 2). Cells in a partitioned flow direction array can be classified according to their location within the flow direction graph and within a partition (Figs. 1 and 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 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. (2019) 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 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

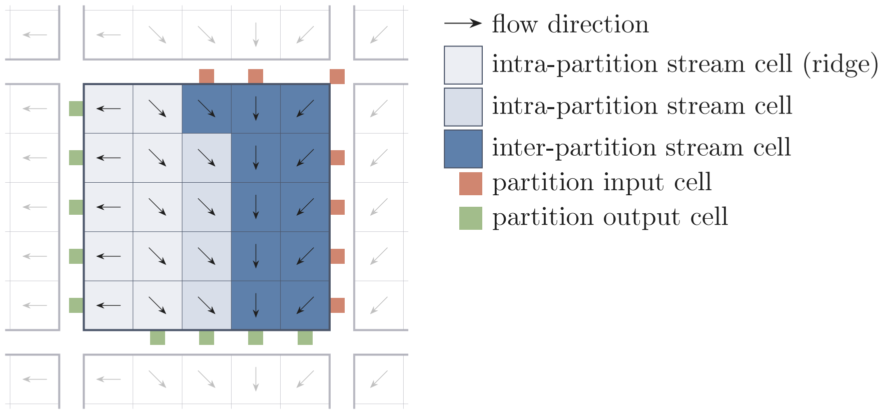

flow direction
intra-partition stream cell (ridge)
intra-partition stream cell
inter-partition stream cell
partition input cell
partition output cell

Fig. 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. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

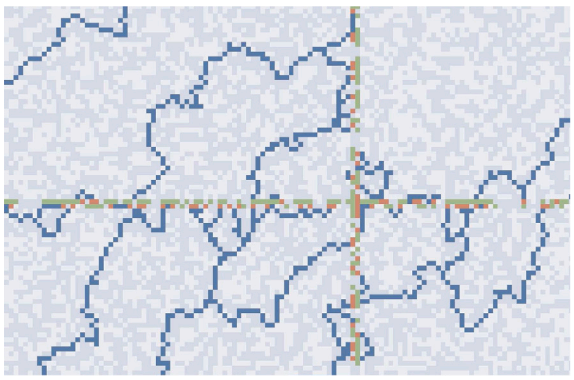


Fig. 2. Flow directions reclassified according to the classes from Fig. 1, for a small area of the MERIT Hydro dataset (Yamazaki et al., 2019). The map shows the borders of four adjacent partitions. Most cells at the borders are partition output cells, producing material to forward to matching partition input cells of a neighbouring partition. Relatively few cells are part of an inter-partition stream; most are intra-partition stream cells.

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. (2019), 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 (Fig. 2).

Compared to the algorithm by Barnes (2017), we have changed the procedure for calculating the results for the inter-partition stream cells. As described in Section 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.

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.

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.

Table 2
Hardware and software platform of cluster nodes used in experiments. All cluster 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	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 (Kaiser et al., 2021)
MPI	Open MPI version 4.0.5

Table 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 centred around a cell in the middle of the continent (Fig. B.5). In case of weak scalability, the array sizes shown are scaled with the number of workers.

	Worker	Dataset	Array size
`accu_threshold`	CPU core	MH1	12 000 × 12 000
	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

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.

Additional details on our algorithms can be found in Appendix A.

4. Experiments

We performed various experiments to characterize the performance, scalability and composability of our new flow accumulation algorithms. In all experiments, we used the MERIT Hydro dataset (Yamazaki et al., 2019) 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 × 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 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.

4.1. Algorithm

The algorithms described in Section 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 4.3). We performed the experiment on the same dataset as used in the weak scaling experiment over CPU cores of `accu_threshold` (Table 3) when using 6 CPU cores. This array has 30 000 × 30 000 cells and contains relatively few no-data cells. The results of this experiment are given and analysed in Section 5.1.

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 (Karssenberg et al., 2010). We compared the latencies of a single `accu` and a single `accu_threshold` call for the southern half of the African continent (56 059 × 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 (Lana-Renault and Karssenberg, 2013) and the PCR-GLOBWB global water balance model (Sutanudjaja et al., 2017). 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 characterize 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 (Eq. (1)). To characterize 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 (Eq. (2)).

$$\mathrm{RSE}_{strong} = \frac{T_{S,1}}{P \times T_{S,P}} \times 100\% \quad (1)$$

$$\mathrm{RSE}_{weak} = \frac{T_{W,1}}{T_{W,P}} \times 100\% \quad (2)$$

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. We used subsets of the MERIT Africa dataset and a resampled version thereof in the scalability experiments (Table 3, Appendix B). Since the size of the tasks depends on the size of the array partitions, and not every task size results in good performance (Section 2), before performing the scalability experiments we first determined good partition sizes to use Appendix C.

4.3. Composability

In order to characterize the composability of the flow accumulation operations, we used a model containing two calls to a flow accumulation operation (Listing 1). In an actual model, the first one could simulate water transport, while the second one uses the outflow result 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 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 3) when using 8 NUMA nodes. This array has 85 000 × 85 000 cells and contains relatively few no-data cells. We executed each model variant 10 times and selected the smallest latencies.

Listing 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])
```

5. Results

5.1. Algorithm

The trace shows which flow accumulation tasks are executing over time (Fig. 3(a)). 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 3). Concurrent tasks, operating on different partitions, get scheduled in parallel.

5.2. Performance and scalability

Calculating a result for the southern half of the African continent with our new `accu` operation took 1.1 min while PCRaster's version took 3.2 min. Our new `accu_threshold` operation took 1.4 min while PCRaster's version took 3.3 min.

Both the strong and weak scalability of `accu_threshold` over the CPU cores within a NUMA node are higher than 80% (Table 4). 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 (Fig. 4). Additional experiments revealed that the procedure for distributing array partitions over processes contributes to this irregular pattern Appendix D. The speed-up when using 8 NUMA nodes compared to using 1 is about 5.5. In case of the case-study model, strong scaling efficiencies are lower than weak scaling efficiencies. Given the latencies of the model when using a whole cluster node, the scaling efficiencies are high—around 80%. Additional cluster nodes can be used effectively. In case of the strong scalability experiment for the case-study model, we performed an additional experiment, to determine at how many cluster nodes scalability stops and what the maximum associated speed-up is. At 8 cluster nodes, the speed-up is almost 4 and does not increase anymore Appendix E.

Table 4
Relative strong and weak scaling efficiencies.

	Worker	RSE_{strong}	RSE_{weak}
`accu_threshold`	CPU core	83%	84%
	NUMA node	69%	56%
model	CPU core	52%	77%
	NUMA node	48%	67%
	cluster node	73%	84%

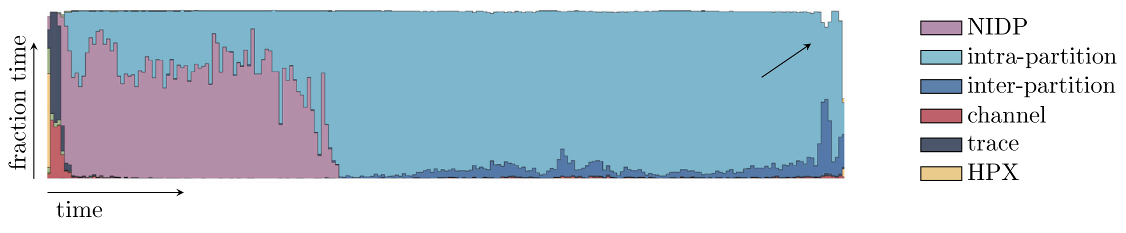
5.3. Composability

Running the model from Listing 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.

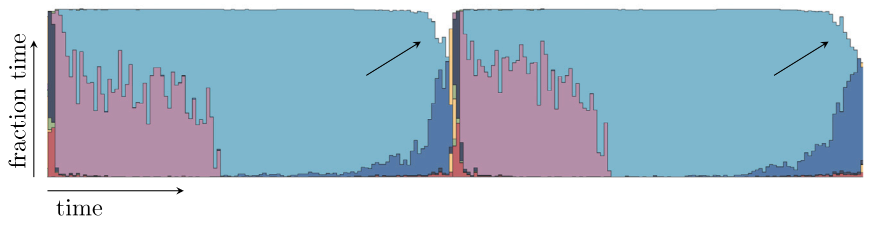
The traces in Fig. 3 illustrate a result of a similar experiment, on a smaller dataset run on the 6 cores within a single NUMA node (Section 4.1). In case of no synchronization point between the operations, even though there is a data dependency between the two operations, tasks from the second call 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 Fig. 3, which results in longer model latencies.

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.

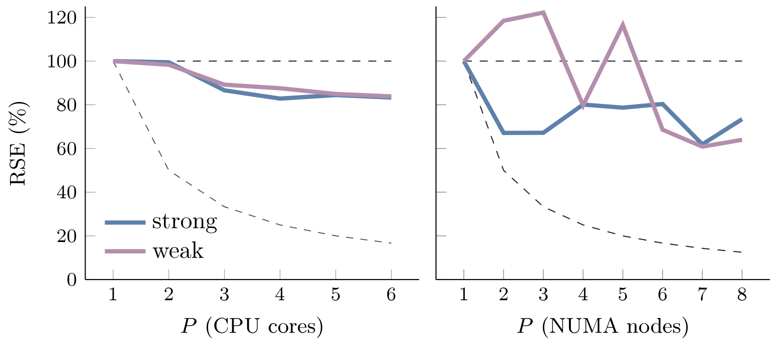


(a)

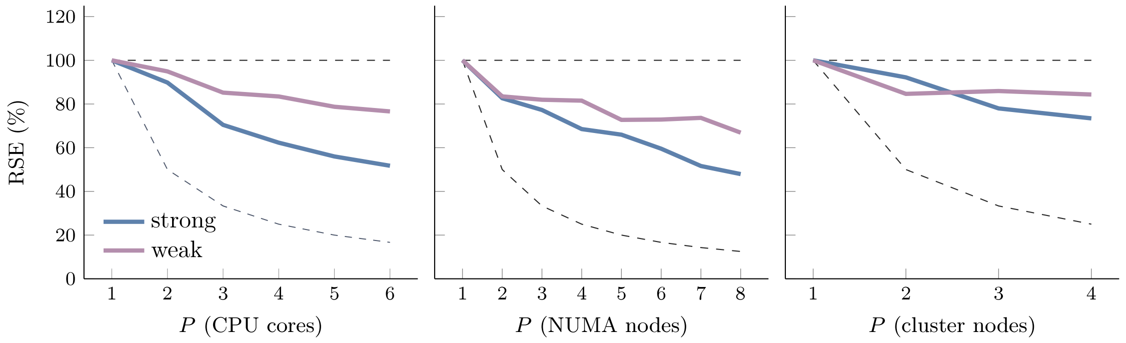


(b)

Fig. 3. Traces of flow accumulation tasks created by two calls to `accu_threshold` (Listing 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 3.3.



(a)



(b)

Fig. 4. Relative scaling efficiencies (RSE) of experiments performed for `accu_threshold` (a) and the case-study model (b, Table 3). For reference, efficiencies for linear scaling (upper dashed line) and serial scaling (lower dashed line) are also shown.

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 Appendix D 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.

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.

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 (de Jong et al., 2021), 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 (Te Chow et al., 1988). 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 extend this increases the performance of models.

CRediT authorship contribution statement

Kor de Jong: Conceptualization, Designed and implemented the framework, Writing - original draft. **Debabrata Panja:** Conceptualization, Providing inputs for writing the manuscript. **Derek Karssenberg:** Conceptualization, Providing inputs for writing the manuscript. **Marc van Kreveld:** Conceptualization, Providing inputs for writing 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).

The authors would like to thank Oliver Schmitz for creating the LUE Conda packages.

Computer code availability

The LUE scalable modelling framework 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 (de Jong and Schmitz, 2021), and containing additional information about the commands used for the described experiments can also be found on GitHub, at https://github.com/computationalgeography/paper_2021_routing.

Appendix A. 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 cell, sink cell, 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 cell, sink cell, 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.
        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.

Table C.5
Partition sizes used in scaling experiments.

	worker	partition size (no. of cells)
`accu_threshold`	CPU core	2 500 × 2 500 (6.25×10^6)
	NUMA node	2 500 × 2 500 (6.25×10^6)
model	CPU core	1 500 × 1 500 (2.25×10^6)
	NUMA node	1 500 × 1 500 (2.25×10^6)
	cluster node	2 500 × 2 500 (6.25×10^6)

Appendix B. Data

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. Fig. B.5 shows the bounding boxes used for the weak scalability experiments.

Appendix C. Partition size

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 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 C.5 and based on the distribution of latencies over various partition sizes (Fig. C.6). 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.

Appendix D. Irregular scaling efficiencies

The scalability experiments of the `accu_threshold` operation over NUMA nodes resulted in an irregular pattern of scaling efficiencies (Fig. 4). To gain insights into why 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

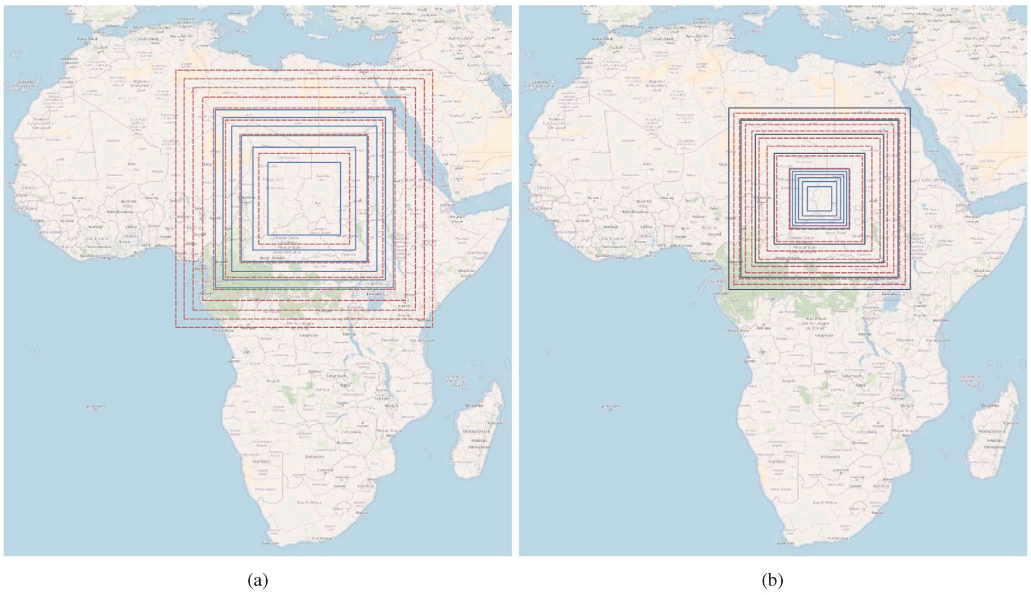


Fig. B.5. Bounding boxes of arrays used for weak scalability experiments for `accu_threshold` (B.5(a)) and the case-study model (B.5(b)). In solid blue the 6 areas used for scaling over the CPU cores within a NUMA node. In dashed red the 8 areas used for scaling over the NUMA nodes within a cluster node. In solid grey the 4 areas used for scaling over the cluster nodes within a cluster partition. (Base map and data from OpenStreetMap and OpenStreetMap Foundation.). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

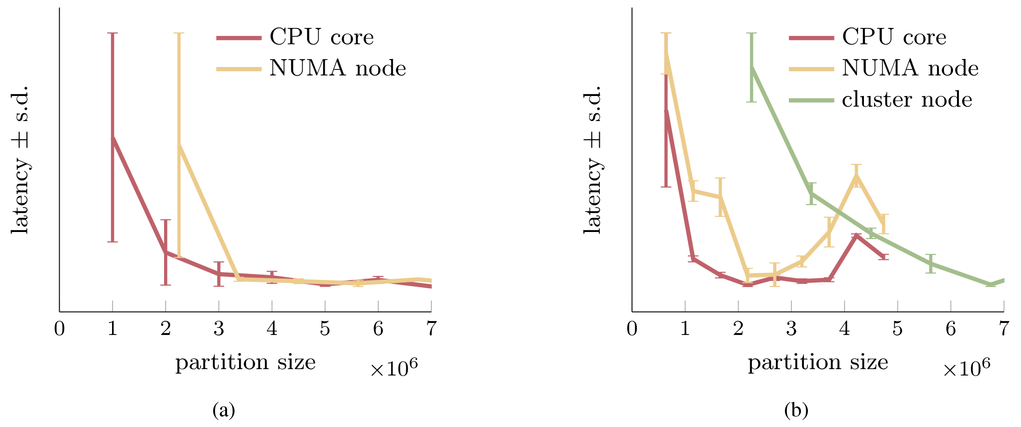


Fig. C.6. For the scaling experiment for `accu_threshold` (C.6(a)) and the case-study model (C.6(b)), the latencies of running the experiment for a range of partition sizes (number of cells) on the maximum number of workers and maximum array size. The actual latencies are different for the different kinds of workers. The y-axis starts at zero and increases linearly.

the same strong and weak scalability experiments as presented in Section 5.2, but for a different part of the African continent (centred 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 (Fig. D.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 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 (Gotsman and Lindenbaum, 1996). 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 Section 5.2, 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 (Fig. D.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 where 16% slower (9455 ms versus 8153 ms).

Appendix E. Maximum speed-up

As described in Appendix B, the number of workers over which scalability experiments can be performed is limited by the availability

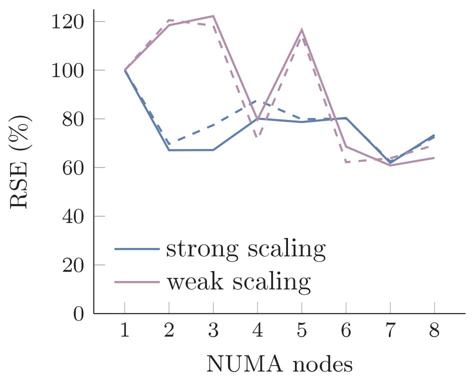


Fig. D.7. Relative strong and weak scaling efficiencies (RSE) of experiments performed for `accu_threshold`. The solid lines correspond with those from the original experiments shown in Fig. 4(a), and the dashed lines with results of the same experiments, but for a different flow direction field.

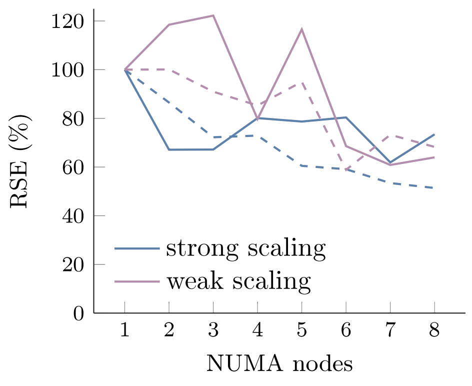


Fig. D.8. Relative strong and weak scaling efficiencies (RSE) of experiments performed for `accu_threshold`. The solid lines correspond with those from the original experiments shown in Fig. 4(a), and the dashed lines with results of the same experiments, but using a different procedure for assigning partitions to processes.

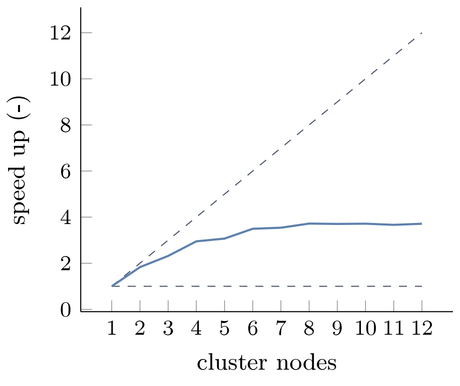


Fig. E.9. Speed-ups for the strong scaling experiment performed for the case-study model (Table 3). For reference, speed-ups for linear scaling (upper dashed line) and serial scaling (lower dashed line) are also shown.

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 a larger number of cluster nodes than discussed in Section 5.2. At 8 cluster nodes, the speed-up is almost 4 and does not increase anymore (Fig. E.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.

References

Barnes, R., 2017. Parallel non-divergent flow accumulation for trillion cell digital elevation models on desktops or clusters. Environ. Model. Softw. 92, 202–212. http://dx.doi.org/10.1016/j.envsoft.2017.02.022.

Burek, P., van der Knijff, J., de Roo, A., 2013. LISFLOOD – Distributed water balance and flood simulation model – Revised user manual. http://dx.doi.org/10.2788/24982.

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

Cordonnier, G., Bovy, B., Braun, J., 2019. A versatile, linear complexity algorithm for flow routing in topographies with depressions. Earth Surf. Dyn. 7, 549–562. http://dx.doi.org/10.5194/esurf-7-549-2019.

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

Esri, 2021. ArcGIS desktop. https://www.esri.com.

GDAL/OGR contributors, 2021. GDAL/OGR Geospatial Data Abstraction Software Library. Open Source Geospatial Foundation, https://gdal.org.

Gotsman, C., Lindenbaum, M., 1996. On the metric properties of discrete space-filling curves. IEEE Trans. Image Process. 5, 794–797. http://dx.doi.org/10.1109/83.499920.

Hendriks, M., 2010. Physical Hydrology, first ed. Oxford University Press.

de Jong, K., Panja, D., van Kreveld, M., Karssenberg, D., 2021. An environmental modelling framework based on asynchronous many-tasks: Scalability and usability. Environ. Model. Softw. 139, 104998. http://dx.doi.org/10.1016/j.envsoft.2021.104998.

de Jong, K., Schmitz, O., 2021. Computationalgeography/lue: LUE-0.3.0: Scientific database and environmental modelling framework. http://dx.doi.org/10.5281/zenodo.5535686.

Kaiser, H., Diehl, P., Lemoine, A.S., Lelbach, B.A., Amini, P., Bergé, 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., 2020. HPX - The C++ standard library for parallelism and concurrency. J. Open Source Softw. 5, 2352. http://dx.doi.org/10.21105/joss.02352.

Kaiser, H., Simberg, M., Lelbach, B.A., Heller, T., Bergé, A., Biddiscombe, J., R., A., Bikineev, A., Mercer, G., Schäfer, A., Huck, K., Lemoine, A.S., Kwon, T., Habraken, J., Nair, A., Anderson, M., Brandt, S.R., Copik, M., srinivasyadav18, Finomnis, Bourgeois, D., Blank, D., Gonidelis, G., Gupta, N., rstobaugh, Jakobovits, S., Amatya, V., Viklund, L., Diehl, P., Khatami, Z., 2021. STEllAR-GROUP/hpx: HPX V1.7.1: The C++ standards library for parallelism and concurrency. http://dx.doi.org/10.5281/zenodo.5185328.

Karssenberg, D., 2006. Upscaling of saturated conductivity for Hortonian runoff modelling. Adv. Water Resour. 29, 735–759. http://dx.doi.org/10.1016/j.advwatres.2005.06.012.

Karssenberg, D., Schmitz, O., Salamon, P., de Jong, K., Bierkens, M.F.P., 2010. A software framework for construction of process-based stochastic spatio-temporal models and data assimilation. Environ. Model. Softw. 25, 489–502. http://dx.doi.org/10.1016/j.envsoft.2009.10.004.

Kotyra, B., Chabudziński, Ł., Stpiczyński, P., 2021. High-performance parallel implementations of flow accumulation algorithms for multicore architectures. Comput. Geosci. 151, 104741. http://dx.doi.org/10.1016/j.cageo.2021.104741.

Lana-Renault, N., Karssenberg, D., 2013. PyCatch: Component based hydrological catchment modelling. Cuad. Investig. Geogr. 39, 315–333. http://dx.doi.org/10.18172/cig.1993.

Neteler, M., Bowman, M.H., Landa, M., Metz, M., 2012. GRASS GIS: A multi-purpose open source GIS. Environ. Model. Softw. 31, 124–130. http://dx.doi.org/10.1016/j.envsoft.2011.11.014.

O'Callaghan, J.F., Mark, D.M., 1984. The extraction of drainage networks from digital elevation data. Comput. Vis. Graph. Image Process. 28, 323–344. http://dx.doi.org/10.1016/s0734-189x(84)80011-0.

Ortega, L., Rueda, A., 2010. Parallel drainage network computation on CUDA. Comput. Geosci. 36, 171–178. http://dx.doi.org/10.1016/j.cageo.2009.07.005.

Sten, J., Lilja, H., Hyväluoma, J., Westerholm, J., Aspnäs, M., 2016. Parallel flow accumulation algorithms for graphical processing units with application to RUSLE model. Comput. Geosci. 89, 88–95. http://dx.doi.org/10.1016/j.cageo.2016.01.006.

Sutanudjaja, E.H., van Beek, R., Wanders, N., Wada, Y., Bosmans, J.H.C., Drost, N., van der Ent, R.J., de Graaf, I.E.M., Hoch, J.M., de Jong, K., Karssenberg, D., López, P.L., Peßenteiner, S., Schmitz, O., Straatsma, M.W., Vannametee, E., Wisser, D., Bierkens, M.F.P., 2017. PCR-GLOBWB 2: A 5 arc-minute global hydrological and water resources model. Geosci. Model Dev. Discuss. 1–41. http://dx.doi.org/10.5194/gmd-2017-288.

Te Chow, V., Maidment, D.R., Mays, L.W., 1988. Applied Hydrology. In: Civil Engineering, McGraw-Hill.

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

Yamazaki, D., Ikeshima, D., Sosa, J., Bates, P.D., Allen, G.H., Pavelsky, T.M., 2019. MERIT Hydro: A high-resolution global hydrography map based on latest topography dataset. Water Resour. Res. 55, 5053–5073. http://dx.doi.org/10.1029/2019wr024873.

Zhou, G., Wei, H., Fu, S., 2019. A fast and simple algorithm for calculating flow accumulation matrices from raster digital elevation. Front. Earth Sci. 13, 317–326. http://dx.doi.org/10.1007/s11707-018-0725-9.