Computers and Electronics in Agriculture 236 (2025) 110392

Contents lists available at ScienceDirect

Computers and Electronics in Agriculture

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

Original papers

Efficient and scalable crop growth simulations using standard big data and distributed computing technologies

Rob Knapen [a,*], Allard de Wit [a], Eliya Buyukkaya [a], Petros Petrou [c], Dilli Paudel [b], Sander Janssen [a], Ioannis Athanasiadis [b]

[a] Wageningen Environmental Research, Wageningen University & Research, PO Box 47, 6700 AA Wageningen, The Netherlands
[b] Artificial Intelligence Group, Wageningen University & Research, The Netherlands
[c] UBITECH, Thessalias 8 & Etolias, Chalandri 15231, Greece

ARTICLE INFO

Keywords:
Distributed computing
Apache Spark
Crop yield forecasting
WOFOST crop growth model
Benchmarking
Kubernetes
HPC

ABSTRACT

The digitization in agriculture has led to an explosion of highly detailed data generated, offering opportunities for further optimizing resource use in food production systems. However, managing and processing these growing data volumes presents significant challenges. This study investigates the suitability of standard big data and distributed computing technologies with a crop yield forecasting case study, and benchmarks performance and scalability of storage and compute. To that end a prototype system leveraging the Apache Spark big data analytics framework and using the WISS-WOFOST crop growth simulation model is assembled and evaluated for its efficiency and scalability when running large numbers of simulations using distributed computing on commonly available infrastructure. Existing data for maize and winter wheat, as typical summer and winter crops, is prepared for distributed storage and processing and used to measure the performance of the system on clusters of increasing sizes, from small Kubernetes Cloud deployments to large HPC configurations. Specific attention is paid to the aggregation of the grid-based simulation results to larger administrative regions for follow-up analysis and reporting. Our results demonstrate that the selected standard big data and distributed computing technology simplifies the application of distributed processing and storage, making the related trade-off between runtime and costs more attainable. By increasing the distribution of our system 64 times and the total number of cores used 45 times compared to the baseline, we obtained a 99% reduction in simulation processing time and a 95% decrease in the aggregation time of the simulation results, making detailed forecasting for large areas more tractable. However, distributed implementations remain inherently more complex than conventional ones. As such, the construction and use of distributed systems will continue to be a challenge for agricultural agronomists and agricultural data scientists.

1. Introduction

The ongoing digitization of agriculture provides increasing amounts of data that can be used to further improve our food production systems, including optimizing resource use on the farm with precision agriculture, forecasting regional and global yields, helping us to adapt to the effects of climate change, and allowing consumers to make better informed decisions about their food purchases by tracking and providing sufficient information (Parra-López et al., 2024, Wolfert et al., 2017). All of these require attention to how increasing amounts of data are stored, managed, and processed, both in operational systems and for research purposes.

Higher demands for data processing and storage capacity can be handled in the first place by upgrading computing equipment with more storage space, more memory, and faster processors. This is referred to by the term "scaling up". To make the most efficient use of the available hardware, concurrent processing can be implemented, utilizing all CPU cores in the system to their maximum. However, eventually all this will reach system limits, and the use of a form of distributed computing (known as "scaling out") has to be considered in order to still be able to perform all required processing in a timely fashion (see Hennessy and Patterson, 2011).

Distributed computing refers to the use of a cluster of computers, typically with highly available resources (processing cores, memory, and disk space). The main computational task can then be divided and solved using all computers available in such a cluster, with a final task that collects all outputs and integrates them to produce the end

* Corresponding author.
E-mail address: rob.knapen@wur.nl (R. Knapen).

https://doi.org/10.1016/j.compag.2025.110392
Received 10 March 2024; Received in revised form 4 April 2025; Accepted 6 April 2025
Available online 2 May 2025
0168-1699/© 2025 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

result. Due to the inherent higher complexity of distributed systems, the initial (good) tendency usually is to avoid using such solutions, stick to familiar single computers, and attempt to fit the computational job. Or, not being familiar with the existing available IT technologies, bespoke systems using multiple dedicated computers are constructed (such as, for example, in Kim et al., 2020) which are highly dependent on specialized software and tailored servers, usually not very fault tolerant and potentially a maintenance nightmare.

Fortunately, custom individually managed servers that require personal attention can now be replaced by ephemeral commodity servers either located on-premises, for example, as part of local Kubernetes cluster (http://kubernetes.io) or HPC (high-performance computing) facilities, or hosted remotely ('in the Cloud'), e.g. Microsoft Azure, Amazon Web Services, or the Google Cloud Platform. Making use of such computing facilities still requires breaking down the total computational workload into a number of smaller tasks that can then be processed in parallel. This scheduling, or orchestration, software can be custom developed, for example, by accessing the Kubernetes Control Plane that allows the dynamic creation and deletion of nodes in the cluster (if administrators allow) as done by Kim et al., 2021, or by programming master–worker node implementations using internal networking and a central database for distribution of tasks, as described in Li et al., 2023. Alternatively, standard big data and distributed computing technologies can be leveraged, such as the currently well-known open source frameworks Dask (http://dask.org), Ray (http://ray.io), and Apache Spark (http://spark.apache.org).

In this paper, we investigate whether such a standard distributed computing solution can be successfully applied to a classical use case from the agricultural domain, namely that of crop yield forecasting. This is a key component of crop yield monitoring systems, which are important tools for agricultural monitoring (e.g. for anomaly detection and early warning) (Fritz et al., 2019). They are critical to informing stakeholders on the current outlook on crop production and provide support for policies on market intervention and import/export regulations. A few examples are the international Agricultural Market Information System (AMIS, http://amis-outlook.org) of the Food and Agriculture Organization (FAO), and the GeoGLAM Crop Monitor (https://cropmonitoring.org). A variety of crop yield monitoring systems exist, some based on field visits and in situ observations (e.g. USDA system U.S. Department of Agriculture, 2012), some relying on remote sensing imagery (e.g. FEWS-NET Ross et al., 2009, NASA Harvest Whitcraft et al., 2020, CHARMS Huang et al., 2018) and a final category applying deterministic crop growth models, often combined with other data sources including remote sensing data (e.g. EC MARS Lecerf et al., 2019). This last category of crop yield monitoring solutions ingests data on crop, soil and agro-management, and historical weather data, as well as ensemble weather forecasts. Next, a cropping systems model is applied to provide estimates of various crop variables, such as phenology and biomass, and to provide a forecast of the expected crop yield at the end of the season. Monitoring systems are traditionally implemented on a consistent geographical grid of 50 × 50 km or 25 × 25 km, on which all data are prepared and different information layers are intersected. With the advance of digitization in agriculture, more data is becoming available with increasingly detailed spatial resolution. Examples of such data products are the AgERA5 weather data set (Boogaard and van der Grijn) at a resolution of 0.1 degrees, and the SoilGrids soil database at a spatial resolution of 1 km (Poggio et al., 2021). Crop monitoring systems would be more useful if they were upgraded to provide outputs with higher spatial resolution utilizing those new data sets (Paudel et al., 2023). This increases their usefulness as the outputs become relevant for different user groups which often require data on cropping conditions for smaller spatial entities (e.g., watersheds and counties). For example, the outputs of such systems could be used for index-based insurance (Afshar et al., 2021) and real-time advisory services for farmers and extension services (Hack-ten Broeke et al., 2019). However, such an upgrade to high resolution significantly increases the computational requirements for operational systems as the number of unique simulation units grows. How to address this, preferably with existing tools, is an important research question.

Thus, in this study, we investigated, as the main research objective, *whether a commonly used distributed computing framework could successfully be applied to run simulations at scale using a numerical crop simulation model, while benchmarking both performance and scalability of the system on data sets for two types of crops (maize and winter-wheat, as typical summer and winter crops), and deploying it on computer clusters of various sizes, using both Cloud and HPC configurations*. However, to reach this objective, we first needed to look at an existing system and adapt its key data management and processing steps to make use of distributed technologies. Based on our familiarity with the WOrld FOod STudies (WOFOST) cropping system model and its operational use in the MARS Crop Yield Forecasting System (MCYFS) (de Wit et al., 2019), these were selected as the basis for this work.

In Section 2 we describe an implementation of a distributed system that embeds a WOFOST cropping system model as a data transformation in Apache Spark, making use of aspects of declarative (functional) programming. This includes key aspects for data management in distributed computing, such as the data input and output schemas, data denormalization, data serialization, and data deserialization, what distributed computing entails, and how we used it to run the existing crop model, as well as for aggregating the outputs of all model runs. Having this prototype system in place, we used it to run the crop simulations for the maize and winter-wheat data sets, on various distributed computing configurations. Section 3 documents these benchmark experiments and their results, which we further discuss in Section 4, with the final conclusions in Section 5.

2. Methodology

2.1. Overview

To meet the research objective presented in the Introduction, we chose to use the gridded implementation of the WOFOST cropping system model (de Wit et al., 2020b) within the European MARS crop monitoring system (van der Velde et al., 2019) as a reference for prototyping and benchmarking a distributed technologies-based system as a case study.

Fig. 1 shows an overview of the processing steps in the prototype system and already mentions some of the distributed technologies used in the prototype (such as the Apache Spark framework), which will be described in detail later. As part of the implementation, we needed to find effective solutions for (i) managing the large input data for crop growth simulations using distributed storage technologies (box 1 in Fig. 1); (ii) efficiently running a numerical crop growth model on a computer cluster (box 2 in the same figure); (iii) collecting the results of large numbers of both successful and failed simulations from all computers in the cluster (also in box 2); (iv) applying post-processing operations on the vast amounts of simulation outputs to obtain the final results (box 3). More details about these technologies and the solutions chosen are described in the following subsections.

Our prototype leaves out or simplifies steps at the boundary of the system, i.e. it uses data from the existing MARS system, and has no user interface components for control and visualizing outputs. To construct an operational system, these will have to be added; however, they are not needed to experiment with the processing of the core crop simulation model and to perform benchmark measurements.

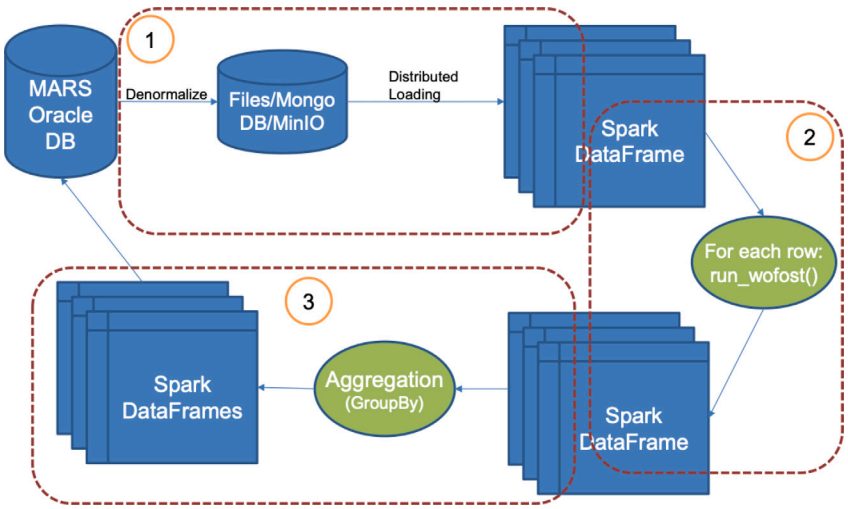

Fig. 1. Overview of the crop simulation processing steps. (1) Extraction, transformation (denormalization), and loading of data into the Spark analytical framework, which represents it as a `DataFrame`. (2) Distributed and parallel running of crop simulations for each row of the `DataFrame`, producing results as a new `DataFrame`. (3) Using Spark SQL commands to calculate aggregated outputs.

2.2. Case study

A clear use case for the sketched prototype that can scale out crop simulations is the computation and use of the resulting crop simulation outputs for regional crop yield forecasting. Examples of such systems are the mentioned European MARS Crop Yield Forecasting System (MCYFS) (van der Velde et al., 2019) and the CRAFT system (Vakhtang et al., 2019) which were both designed to provide estimates of crop production at regional level during the cropping season. Similar systems are in place in other parts of the world, such as CGMS-Morocco (de Wit et al., 2013; Lahlou).

Complex systems like MCYFS or CRAFT typically segment the spatial domain into small spatial units for which all input variables are assumed to be homogeneous. Next, a crop simulation model is applied to each spatial unit, and its output is collected. The reason for this approach is the non-linear response of a crop model to its inputs which implies that simulations must be done at the lowest spatial level, followed by aggregating the model outputs toward higher levels such as grids or regions.

Currently, the MCYFS operates at a spatial resolution determined by the intersection of the weather grid (25 × 25 km) and the soil map, which provides homogeneous grid/soil units. For a crop such as winter wheat, the system employs 150,000 individual soil/grid combinations that have to be simulated individually. Given that the system simulates about 20 types of crops operationally, this will add up to approximately 3 million individual simulations with the included model (WOFOST, further described in Section 2.3). These simulations must be repeated every 10 days to take into account the latest weather conditions.

Besides the computational burden that comes with running WOFOST on small individual units, handling results from all those individual units also requires quite some attention. The WOFOST simulation results at the lowest level are of little use for yield forecasting and visualization. Therefore, results are aggregated to grid and regional levels, each following a different aggregation scheme. Aggregation from the lowest level toward grid level is performed by computing an average of the simulated variables weighted on the relative area of each soil type within the grid. Aggregation of grid level results toward the lowest level regions is performed using the area of arable land for each grid as a proxy for crop area, while aggregations from the lowest level toward higher levels are carried out by crop area estimates obtained from EUROSTAT (http://ec.europa.eu/eurostat). Moreover, at each level of aggregation, a climatology is required that is used to produce maps and charts showing the current conditions relative to the long-term statistics.

In the current implementation of MCYFS, all crop simulations are done on a single compute server that has a multi-core processor. The source code and data retrieval of the model have been optimized for performance on this limited infrastructure. The current distribution of simulations across the processor cores is done by splitting the spatial domain into tiles, which works in practice but gives little flexibility: tiles which consist of fewer simulation units (e.g., which contain few crop areas) will finish quickly but cannot pick up tasks from other tiles (i.e., there is no 'work-stealing' between tasks implemented). In addition, results from individual crop simulations are first written to files and then loaded into a relational database using dedicated data loading tools, which takes a considerable amount of time. Finally, the aggregation of simulation results is carried out using database procedures that compute the weighted averages for grids and region, which is demanding, although something that databases excel at. Finding solutions for the challenges mentioned above will be critical for scaling crop monitoring systems toward higher spatial resolutions.

2.3. Embedding a WOFOST model

The WOFOST cropping system model has been used in the MARS crop yield forecasting system since the early 1990s (de Wit et al., 2019). It is a core component used to determine the impact of weather conditions on crop growth for the major arable crops in Europe and other areas of the world. The model computes crop growth and development in a biophysical and process-based way and summarizes the status of the crop through a set of core state variables: phenological development, biomass in various plant organs, leaf area of the plant canopy and its interaction with the soil through soil moisture. The changes of these variables from day to day are computed based on the underlying processes such as photosynthesis, respiration, leaf dynamics, evapotranspiration and how they are influenced by weather.

Crop growth simulation models themselves, such as those used in this study, are difficult to further parallelize efficiently because of their internal use of interrelated state variables calculated in time steps. Moreover, many of such existing models were not developed with parallelization and distributed computing in mind and contain legacy programming and design constructs that make a switch difficult. For example, Jang et al. (2019) developed a parallel computing framework to run a spatialized version of the EPIC model. However, their approach still relies on file-based input/output which severely limits distribution over multiple computing nodes. In addition, the structure of the model is such that simulations are limited to using multiple CPU cores on a

single computer, rather than using all CPU cores in a cluster of multiple computers. Similarly, Alderman (2021) developed a gridded version of the DSSAT-CSM model that has comparable limitations on scalability.

Running a single WOFOST simulation for a given location, crop, and year can be carried out in less than 100 ms using an efficient implementation on modern computer hardware. However, the MARS system is a spatial implementation of WOFOST which is computationally demanding overall. Given the strong non-linearity in crop models, a spatial implementation of WOFOST must adhere to the principle of simulation at the lowest level first, followed by performing output aggregations in time and space. Therefore, each unique combination of the weather grid, crop mask, soil map, and agro-management must be simulated separately, which generates a large number of unique simulation units.

Developments within the MARS system have steadily increased computational requirements. These included a decrease in the size of the meteorological grid from 50 × 50 km to 25 × 25 km, extension of the area to be monitored, and inclusion of additional crop types. Moreover, the inclusion of ensemble weather forecasts strongly increases the computational load because model ensembles need to be calculated instead of deterministic model runs. Efficiently handling such a computational load requires an optimized implementation of WOFOST as well as a data infrastructure that can handle a large amount of input data and output data in a performant way.

Recently, the 7.2 version of WOFOST (de Wit et al., 2020b), originally written in FORTRAN, has been reimplemented in the Java programming language, focusing on efficiency, performance, modularity and portability. This edition is also known as WISS-WOFOST and consists of a lightweight framework (WISS, for Wageningen Integrated Systems Simulator) (van Kraalingen et al., 2020), and a component-based implementation of the crop simulation model itself. An important design principle for this edition was that the model should be free of side effects as much as possible and can be run completely in memory. Furthermore, the source code has been fully aligned with the WOFOST implementation, which is part of the Python Crop Simulation Environment (PCSE) (de Wit, 2021).

The only remaining side effects of WISS-WOFOST are exceptions (errors when running the simulation) and log messages. These still make it an *impure function* (see the section on Functional programming) since it is not completely referential transparent, but both types of side effects can be handled with relative ease. This functional programming-style implementation of WOFOST together with the versatility and efficiency of the Java programming language greatly increases the opportunities for using WISS-WOFOST in computational challenges and big data applications. Java programs are ultimately compiled into Java bytecode and executed on a Java Virtual Machine (JVM), which then continuously performs dynamic analysis and optimizations of the running code. Some other programming languages are supported by the JVM as well, and language interoperability is then a given. This is particularly convenient for embedding WISS-WOFOST into a framework such as Apache Spark, which is written in Scala, one of the other JVM languages. However, the interoperability of Spark is by no means limited to all JVM languages. It also supports the frequently used programming languages in data science, Python, and R, and it includes functionality to access command-line based models and applications (e.g. compiled C++ or Fortran code) via Linux pipes and interprocess communication.

The input data required by the spatial WOFOST implementation used in the MARS crop yield forecasting system is stored in a relational database management system (RDBMS). The use of a RDBMS is necessary because of the normalized structure and the many relationships that exist between the different input data types (crop, soil, weather) and the WOFOST output data for each unique combination. However, over the course of three decades, the database schema behind the spatial WOFOST model has grown in complexity to accommodate new features and options. This complexity has a negative impact on the database performance for supplying the input data for WOFOST. Typically, many tables have to be traversed and joined in order to obtain all the required inputs for a WOFOST crop simulation, particularly those related to model parameters and crop calendars. Moreover, a RDMBS and the server it runs on can only handle a limited number of simultaneous connections, making them less scalable.

Fortunately, despite its complexity, most of the input data in the spatial WOFOST RDMBS are static: they do not change during the course of a cropping season. Only weather observations are appended based on new daily observations. This provides an excellent opportunity to convert the WOFOST input data stored in the RDBMS into a temporary format that is more suitable for distributed processing. Further on, we will describe the approach we implemented for this.

Finally, the WOFOST model produces a time series of output variables for different model scenarios (e.g., potential or water-limited production) for all unique combinations of grid, soil, and crop. However, in practice, analysts never use the WOFOST results at the lowest level of simulations. Instead, aggregated values are preferred at the grid or regional level because they are easier to handle and visualize. Currently, all WOFOST simulation results at the lowest level are loaded into the RDBMS and the spatial aggregations are carried out using SQL procedures. This step is time-consuming because all data have to be loaded first before aggregation can start. We have therefore experimented with an alternative approach where aggregation of results is already carried out on the Spark `Dataset` that stores the WOFOST simulation output. This results in a much smaller amount of data that have to be loaded into the RDBMS.

2.4. Functional programming

The design of computing systems is closely tied to the programming languages that control them, as each language is built on a computational model that shapes its paradigm and programming style (Backus, 1978). Two major paradigms are imperative programming and declarative programming (Roy and Haridi, 2004).

Imperative programming languages — such as C, Java, and Python — are based on the Von Neumann computer architecture (described in 1945) and operate through sequences of commands that modify the program's state. While modern imperative languages have begun incorporating functional features, their reliance on mutable state and sequential execution limits abstraction and composition.

In contrast, declarative programming languages — including *functional programming languages* such as Erlang, Haskell, and OCaml — are rooted in mathematics and lambda calculus (Rosser, 1941). They follow a declarative approach, constructing programs from pure, referentially transparent functions that avoid side effects and mutable state. Features like immutability, higher-order functions, and recursion make functional programming well-suited for concurrent and distributed computing, as pure functions are easier to parallelize and reason about.

Despite these advantages, functional programming can present challenges, including a steeper learning curve due to its abstract nature, potential inefficiencies in CPU and memory usage, and longer compile times due to advanced type checking and code generation. However, these trade-offs often lead to more robust, maintainable, and reliable software systems.

2.5. Using distributed data storage

Distributed computing is frequently used for data-intensive applications, which requires efficient data storage technologies to prevent persistence-related bottlenecks. While traditional relational databases excel in structured data management due to their robust research foundation and optimizations, they often struggle to meet the scalability and flexibility demands of big data. To address these limitations, NoSQL databases have emerged, offering benefits such as horizontal scalability,

schema flexibility, fault tolerance, and high availability—often at the cost of delayed consistency.

Unlike traditional databases evaluated on ACID properties (Atomicity, Consistency, Isolation, Durability), modern distributed databases are better evaluated using Brewer's CAP theorem (Brewer, 2012). This theorem states that distributed databases must prioritize either Consistency and Partition Tolerance (CP) or Availability and Partition Tolerance (AP). Highly available systems typically adopt eventual consistency to allow for non-blocking synchronization. The main categories of NoSQL include key value stores, column-oriented storage, document stores, and graph databases, with open-source and proprietary implementations available (Siddiqa et al., 2017).

Beyond NoSQL, other big data storage solutions include parallel file systems and object storage systems. Although parallel file systems, such as Lustre (www.lustre.org), offer high performance in high-performance computing (HPC) environments, their strict adherence to POSIX standards limits scalability at very large data volumes. In contrast, object storage systems, such as Amazon S3, Apache Ozone, and MinIO (http://min.io), overcome these bottlenecks by using abstract data containers with immutable objects, stateless operations, and flexible metadata schemas (Liu et al., 2018). Initially suited for write-once, read-many workloads, improvements in performance, and latency have expanded their applicability.

In this study, the initial use of MongoDB (http://mongodb.com), a document-oriented NoSQL database, was chosen for its ability to handle large volumes of JSON documents, which matched the crop simulation input and output data format. However, the system was eventually switched to MinIO object storage due to its native compatibility with the Kubernetes environment, ease of deployment, and administrative simplicity. In practice, Poznan HPC cluster administrators found integrating MinIO to be more straightforward than setting up a network-connected MongoDB installation, highlighting the practical advantages of object storage solutions for scalable data management in distributed systems.

2.5.1. Simulation input data denormalization

Traditionally, the required input data for the crop simulation model are stored and managed using a relational database management system (RDBMS), with a high degree of normalization applied to minimize data redundancy and improve data integrity. As a result, however, collecting all data needed for a single crop simulation then requires a number of table joins or database views, making the data retrieval a relatively slow process. When we want to optimize computing the crop simulations, this retrieval of the input data also has to be taken into account and optimized as well, or at least made suitable for the type of processing that needs to be done. Similar aspects apply to the processing and storage of the output data produced by the crop simulation model. Commonly, the data storage systems used for big data processing accept less data denormalization and actually prefer data duplication when it serves storing the data on multiple computers so that it can be retrieved with higher parallelism by concurrently running processing jobs. Most of the NoSQL database systems described in Section 2.5 operate in such a way.

To transform the highly normalized data in the relational MARS database into a denormalized version, a Data Extractor program was written that creates JSON Lines format output (http://jsonlines.org) with complete simulation input data per line of the file. An abbreviated single input record is shown in listing 1. For readability, it has been expanded across multiple lines; however, in the file it would span only one long line.

These files do get large (multiple gigabytes); however, with a simulation per line they are easy to process (e.g. filter, split, and merge) with standard operating system commands, and they can be significantly compressed for storage or exchange. Big data tools and distributed computing frameworks typically also have the capability to read compressed JSON files directly, although this might be time consuming depending on how well they manage to distribute the total workload.

2.5.2. Data serialization and deserialization

After extracting the data from the RDBMS in denormalized JSON Lines files, they can easily be imported into a MongoDB database (since it is based on working with JSON documents in collections) that can be accessed by Apache Spark, or Spark can read the JSON files directly for processing. In both cases, Spark handles the serialization required and deserialisation from binary format with *Dataset Encoders*. To allow parallel operations in a cluster, Spark handles data via *Resilient Distributed Datasets* (RDD). RDDs are collections of (data) elements partitioned across the nodes of the cluster and that can be operated on in parallel. The *Dataset API* in Spark and the Encoder framework supports the construction of Datasets from JVM objects, and the manipulation of them using functional transformations (such as `map`, `flatMap`, `filter`, and so on). While Datasets are strictly typed, Spark also has a more generic *DataFrame API*, which is in essence a `Dataset` organized in named columns (a `Dataset[Row]`).

For this study, we had the advantage that we could define both the data schema and write the Java JVM objects that match it. By following the `JavaBeans` specification (i.e. make them serializable, ensure a zero-argument constructor, and add accessor methods for all relevant properties), a standard Spark encoder factory (`Encoders.bean(...)`) could be used to create the Encoders from the `JavaBeans`.

The data schema we designed (for details, see Appendix A) consists of a number of thematic blocks (crop, soil, site, agro-management, weather). Every block can have both a set of named parameters and a further flexible list of parameters. The named parameters are usually the key parameters of a block and can be used for grouping or sorting. In the future, these could be used, for example, to optimize the actual number of crop simulations to be performed by running only one simulation for groups that have exactly the same inputs. The flexible list provides the means of holding a variable number of additional parameters. They are stored and retrieved, but are slightly more difficult and time-consuming to operate upon since they require unpacking first (Spark provides the functionality for this).

There is a schema for the input data (`SimulationInput`) shown in Table A.1 and for the output data (`SimulationOutput`) shown in Table A.4. In the input schema, `crop`, `soil`, `site` and `agro-management` follow the same array-based schema containing `name`, `unit` and `value` structure per parameter (Table A.3). Table A.2 represents the schema of `meteo` providing weather data in time series between the start and end dates. On the other hand, in the output schema, `description` gives general information about the simulation (Table A.6) and `message` contains a list of messages generated during simulation run (Table A.4). In addition, `timeseries` provide the simulation results in time series (Table A.7), while `summary` gives an overall summary of the simulation results (Table A.5).

A small caveat, particularly when dealing with data that include geographic names (e.g., countries, regions, or cities), is to ensure that proper character encodings are used, such as `UTF-8`.

2.6. Using distributed computing

Distributed computing refers to the use of what is called a computer cluster with high available resources (mainly cores, memory, and disk space). The computational problem (the 'workload' or job) is typically divided into a number of tasks, and each of those is then solved by one or more computers. Or, in parallel computing jargon, each task is a sequence of instructions that operate together as a group. Tasks are mapped to Units of Execution (UE), which are the concurrently executing entities such as processes or threads. These UEs need to be further mapped to Processing Elements (PE), the actual hardware elements that execute the streams of instructions. The computers in a cluster are connected by a network so that they can communicate, exchange messages and data, and coordinate the work. When needed,

```
{
  "version": 1, "simId": "grid1035126_crop2_variety20095_year1980_smu9030002_stu9000979",
  "simModel": "WOFOST", "simType": "waterlimited",
  "simCrop": 2, "simCropName": "GrainMaize", "simCropVariety":20095,
  "simYear": 1980, "simStartDate": "1980-04-28", "simEndDate": "1980-12-31",
  "location": { "type": "Point", "coordinates": [ 14.49247, 35.86522 ] },
  "sourceType": "CGMS",
  "sourceDetails": { "name": "Malta",
    "grid": 1035126, "smu": 9030002, "stu": 9000979, "altitudeM": 38,
    "gridWeightFactor": 0.111111
  },
  "cropParams": { "id": "crop2_variety20095", "params": [
    { "name": "SPA", "units": [ "[ha.kg-1]" ], "value": [ "0.0" ] },
    "..." ]
  },
  "soilParams": { "id": "stu9000979_smu9030002", "params": [
    { "name": "SMLIM", "units": [ "[cm]" ], "value": [ "0.3173" ] },
    "..." ]
  },
  "siteParams": { "id": "grid1035126_crop2_year1980_stu9000979",  "params": [
    {"name": "ANGSTB", "units": [ "[-]" ], "value": [ "0.42" ] },
    "..." ]
  },
  "agromanagement": { "id": "grid1035126_crop2_year1980", "params": [
    { "name": "IDEM", "units": [ "[date]" ], "value": [ "1980-04-28" ] },
    "..." ]
  },
  "meteo": { "version": 1, "id": "grid1035126",
        "startDate": "1980-04-28", "endDate": "1980-12-31",
    "data": {
      "temperatureMin": [ 11.3, 10.3, "...", 11.3 ],
      "..."
    }
  }
}
```

Listing 1: An abbreviated single sample JSON input record for simulation

there can be a final task that collects all the outputs and integrates or aggregates them to produce the end result.

Computer clusters can be built in various ways, mostly characterized by how memory is shared between computers and how instructions are executed on the data. Flynn's taxonomy (in (Flynn, 1966, 1972)) categorizes the options in SISD (single instruction, single data), SIMD (single instruction, multiple data), MISD (multiple instruction, single data), and MIMD (multiple instruction, multiple data). The latter is the most well-known type, including computational grids, regular High Performance Computing (HPC) facilities, and Kubernetes. HPCs typically are tailored for high performance through the use of high-end components and allow computational jobs to be run by batch processing. That is, a job has to be entered into a queue and will be scheduled to run on the HPC based on the priority and availability of requested resources. Kubernetes environments usually allow more dynamic processing and are more flexible in adding and removing additional resources based on demand. Today, it has become relatively easy to acquire computing resources from cloud providers to use temporarily and pay for them based on usage.

Distributed computing systems rely on middleware software to manage resources and tasks while abstracting low-level hardware details from users. In high-performance computing (HPC) environments, tools like SLURM (http://slurm.schedmd.com) and Torque (http://adaptivecomputing.com/cherry-services/torque-resource-manager/) serve as workload and job managers, often accompanied by modules for pre-configured software libraries or containerized applications, such as those packaged with Singularity (Kurtzer et al., 2017). In contrast, Kubernetes functions as both a platform and middleware, handling the scheduling of Docker (http://docker.io) containers as pods across available nodes and managing additional tasks, including those initiated by applications running within the system.

Modern open source software frameworks such as those mentioned in the Introduction section can help software engineers implement and deploy various kinds of data processing application on such systems. Their main goal is to improve the execution performance by reducing memory usage, disk I/O, and data shuffling based on optimization and tuning techniques. Apache Spark, for example, uses the familiar split-apply-combine programming model and its specialized implementation called Map-Reduce (Dean and Ghemawat, 2008). This model has been heavily used internally by Google (until about 2015). MPI (an implementation of a Message Passing Interface) also uses partitioning and divide-and-conquer techniques to split the processing on the available resources and calculate partial results. While MPI is usually available as a programming library that allows low-level and optimized control over the way distributed processing is performed, the other mentioned frameworks offer medium- to high-level abstraction layers that provide a unified view of the available hardware and handle most of the resource allocation details during data processing. Typically, they support the development and testing of applications on a local computer with small sample data sets, after which the same source code can be deployed and used on a computing cluster to process the full data sets.

Since this use case revolves around performing batches of simulations with the Java based implementation of the WOFOST crop growth model, which will be described later, we choose to build on the Apache Spark framework, which uses Java Virtual Machines (JVM) internally as well making the integration easier. As a side note, Spark also supports the Python (PySpark), R (SparkR), and SQL (spark-sql) programming languages.

2.6.1. Experiments setup

For performance benchmarking, we will compare measurements with the current WOFOST implementation used in MCYFS, the European MARS Crop Yield Forecasting System that has been introduced in Section 2.2. This system runs on Microsoft Windows 10.x, on a server with an Intel Xeon E5 CPU @ 2.3 GHz that has 20 cores in total and 128 GiB memory. In comparison we run the system described in this paper on a physical GNU/Linux (kernel 5.x) server with also an Intel Xeon E5-v1 CPU @ 3.2 GHz, with 12 cores and 40 GiB memory. In addition, on a high-end laptop running macOS 12.x with an Intel Core i9 @ 2.4 GHz, 16 core and 16 GiB memory, and a small Kubernetes (v1.20.7) GNU/Linux (kernel 5.x) cluster with 1, 2, and 6 vCPUs @ 2.4 GHz, allocating 1 core (from an Intel Xeon E3-v2) and 3 GiB memory per vCPU (equal to a worker node in this case). This information is also included in Table 1.

Furthermore, the scalability of the system has also been benchmarked on a Kubernetes (v1.20.7) GNU/Linux (kernel 5.x) cluster, using larger numbers of bigger worker nodes with 4 cores and 16 GiB memory per vCPU @ 2.5 GHz (Intel Xeon E3-v2). The tests were carried out with 1, 2, 4, 8, and 16 worker nodes. Finally, on the Poznan HPC we ran scalability experiments with 32, 48 and 64 worker nodes running GNU/Linux (kernel 5.x) having 7 cores (Intel Xeon E5-v3) and 8 GiB memory each, using the high performance PSNC Eagle cluster.

In all cases Apache Spark version 3.0.2 and the same application (including the same WISS-WOFOST version) were used to execute the crop simulations and aggregate the results. OpenJDK 11 has been used as a JVM with the default garbage collector (GC) selected with no specific tuning done.

2.6.2. Distributing crop simulations with spark

The Spark encoders described above can thus be used to convert the crop simulation input and output data between, e.g. a JSON representation and an instance of a corresponding Java JVM object. As mentioned, the WISS-WOFOST crop simulation model uses both a Java class that holds all input parameters, called `ParXChange`, and a class that collects all the output data produced by the simulation, the `SimXChange` class. WISS-WOFOST has been built so that all needed input can be passed to it via a `ParXChange` instance, which it can then use to run a crop simulation completely in memory. The output of the crop simulation is fully captured in a `SimXChange` instance. Via configuration, it can be set to write log messages e.g. to the console during processing. It is also built to fail fast when an error is detected, terminating with an exception. These can be captured for further handling.

Listing B1 in Appendix B illustrates how only a few lines of code are needed to create the input (`SimulationInput` knows how to represent (part of) its data as a `ParXChange` instance) and output objects, an instance of the model, and use a `TimeDriver` to run a simulation. The `TimeDriver` object in this case externally drives the day-to-day time stepping of the model, until it reaches an end state. During the simulation, the calculated daily states (of the crop and the environment) are recorded in the output object. After completion, these states are used to update the `SimulationOutput` object. Finally, it should be noted that the type of result of the `run` method is a `Try[SimulationOutput]`, so it can be a success with a `SimulationOutput`, or a failure with a Java exception (in Scala `Try` is a type that represents a computation that may either result in an exception, or return a successfully computed value).

The `object` (which in Scala defines a class that has exactly one instance, i.e. a singleton) and its `run` method from the listing B1 are further used as illustrated by the listing B2 (in Appendix B) to perform the crop simulations in a distributed way using Apache Spark. Depending on the hardware on which it is used, Spark takes care of dividing the workload between the available cores and the computers, as illustrated in Fig. 2.

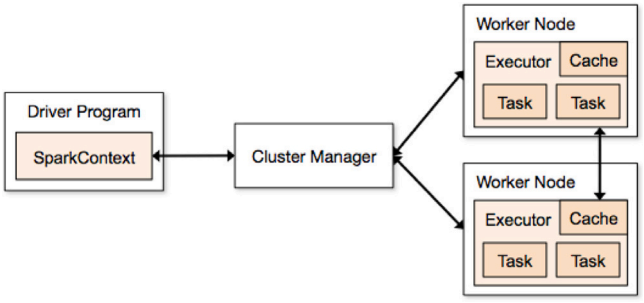

Fig. 2. Main components of Apache Spark, including `Executors` deployed on Worker Nodes allocated from a computer cluster by the Cluster Manager. The Driver Program sets up the `SparkContext` used to control the distributed processing via `Tasks`. (From the Apache Spark documentation).

For that, the main program (called the *driver program* of each Spark application gets a `SparkContext` object that handles the coordination of the processes of the application in a cluster. Via *cluster manager* (e.g. YARN or Kubernetes) which allocates the cluster's resources, it acquires *executors* on *worker nodes*. These are the processes that run the computations and store the data for the application. Next, Spark sends the application code (JAR or Python files) to *executors*, and finally sends them the *tasks* to perform. For their operation, *executors* needs to be able to communicate with each other and with *driver program*. Also, for performance reasons, it is best to run the *driver program* close to *worker nodes*, so in our study we always deployed it on the cluster as well instead of running it on a local computer.

Within a program, the entry point to all Spark functionality is the `SparkSession` class. In listing B2 this is created at the beginning, followed by the construction of the `Encoder` instances for the `SimulationInput` and `SimulationOutput` classes to handle the input data deserialization and output data serialization (see Section 2.5.2). Since Spark is based on *lazy evaluation*, the next lines of code define the expressions to read the simulation input data from JSON file(s), perform crop simulation runs with WISS-WOFOST for all input splits into a requested number of partitions, and write the output of all successful simulations to JSON file(s). Note that it is the final `save` method that will require Spark to actually evaluate all expressions and thus run the simulations. The code also calls the `cache` method on the `Dataset[SimulationOutput]` which holds the results of all crop simulations, so that it can be used for multiple types of (post)processing, including aggregation of the data as described in the next section. The schema of these data is shown in Tables A.4, A.5, A.6, and A.7 in Appendix A.

The Spark `Dataset` (which is a typed version of the more generic Spark `DataFrame`) is a high level view of the partitioned data that represents it as a (very large) table with named columns (similar to a spreadsheet), while hiding all underlying complexity to the user. Typically, the amount of data to be processed makes it impossible to keep everything on a single computer and requires that it be distributed across multiple computers in the cluster. Still, for analytical purposes, Spark will make it appear as a single data set for which a processing pipeline can be specified in a declarative way, which it then translates into a number of possible *logical plans* for execution, of which the best is translated into a *physical execution plan* that is optimized for the available *worker nodes*. Unless it is needed to collect the content of a data set on a single computer (typically where *driver program* runs), it will keep it distributed and apply all requested processing accordingly.

2.6.3. Aggregating crop simulation results with spark

The simulations with WOFOST are carried out for each unique combination of grid and soil unit in order to avoid aggregation errors caused by non-linearity in model responses. For defining these unique

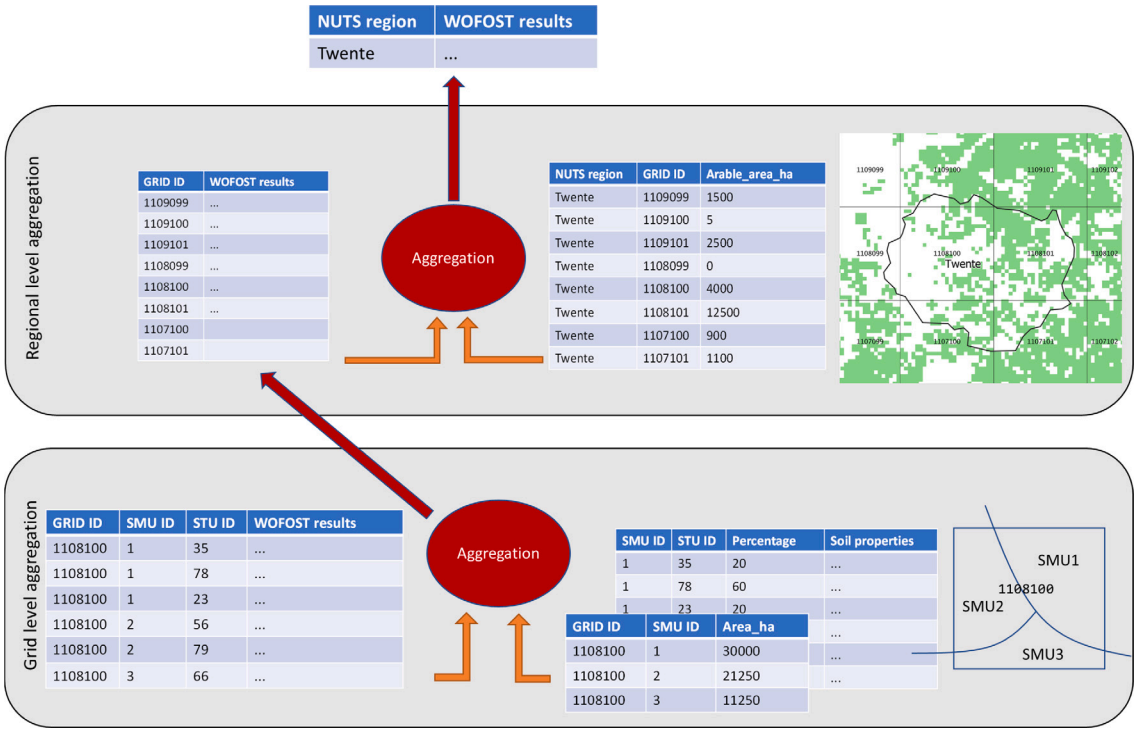

Fig. 3. Schematic representation of the WOFOST result aggregation from individual crop simulations for unique combinations of inter-cell Soil Mapping Unit (SMU) and Soil Typological Unit (STU) to grid cell level (lower gray box) and from grid cells to regional level (upper gray box).

units, the European soil database is intersected with the simulation grid, leading to combinations of grid and so-called *Soil Mapping Units* (SMUs). However, the physical properties of the soil required for running the WOFOST soil water balance are not defined at the SMU level but are available for the so-called *Soil Typological Units* (STU). These STUs are not defined spatially, but only their percentage coverage of an SMU is known. Therefore, aggregating WOFOST simulation results requires one to take into account the interdependency of grid, SMU and STU. Moreover, final output is also required at the regional level, which involves aggregating from grids toward the lower administrative districts (the so-called *NUTS3 regions*) and to the national level (NUTS0). Furthermore, it should be noted that crop areas are not known at the level of grids or SMUs, only for the lowest administrative levels crop-specific area statistics are available.

The aggregation approach is visualized in Fig. 3. At the lowest level, simulation results for each grid, SMU and STU combination can be aggregated toward the grid level based on a weighting factor computed from the configuration of STUs within the SMU and the area of each SMU with the grid. Then, aggregation was performed to the lowest administrative level taking the grid-level results and using the arable land area derived from a land cover map within each grid as a weight factor for the regional results. Finally, aggregation from the lowest administrative level toward higher levels was done by crop area statistics which are available from EUROSTAT.

Despite the complexity of the aggregation scheme, the weight factors that are used to aggregate the results are static and can be computed a priori. Thus, they can be provided as an input to the simulation (e.g. in the JSON Lines formatted files) and replicated in the output from WOFOST. The aggregate of WOFOST results to the grid/region level can thus be carried out with Spark by grouping on the grid/region ID, multiplying the WOFOST output by the respective weight factor, summing up the results, and writing the output to a new Spark `Dataset`. Aggregation of results from the lowest administrative level toward higher levels was not done from within Spark because the weight factors vary by year (e.g., annual crop area statistics). Moreover, the two aggregation steps already reduce the size to such an extent that adding this final step does not provide a substantial reduction of processing time.

Technically, aggregation is implemented in Spark using its select, group, and aggregation methods available for `Datasets`. It could also have been done by using its SQL API, but the total query needed is rather complex, and writing it in programming code makes it more manageable. The complete code is included as listing B3 in Appendix B.

Finally, listing 2 shows a sample output record in JSON format, with the daily time series abbreviated. `TAGP` indicates the *Total Above Ground Production* (dead and living plant organs) in kg/ha, and `TWSO` the *Total Dry Weight of Storage Organs* (dead and living) in kg/ha. More information about these variables can be found in the WOFOST system description (de Wit et al., 2020a). The values are the aggregated results of the water-limited crop simulations for maize, at the indicated grid location.

2.6.4. Deploying on distributed computing infrastructures

Since Apache Spark can be used with various hardware configurations, it allows us to deploy and test the distributed crop simulations system on several setups. The simplest of them is a laptop, desktop computer, or single server. This can be used to develop the software and to run crop simulations on a limited scale (less than 1 million, depending on the hardware). Spark can use all available CPU cores for processing, so it can perform reasonably well on small amounts of crop simulations. It does, of course, introduce some overhead, so a single-computer setup is not necessarily the best approach in this case. In addition, it will not work for data sets that are too large to fit in the memory of the computer. This will require Spark to start overflowing data to temporary files, which reduces the overall performance.

To be able to process larger numbers (over 1 million) of crop simulations within acceptable time frames it is needed to use Spark with a configuration of multiple computers, such as a compute cluster. At the high end, these computers can be nodes of a supercomputer that provides High Performance Computing (HPC) facilities. Spark can be deployed on a subset of these nodes and then used to run the

```
{
    "ERR": 0,
    "Total_Weights": 1.0,
    "NonErr_Weights": 1.0,
    "Crop_Code": 2,
    "Crop_Name": "GrainMaize",
    "Crop_Variety": 20102,
    "Grid": 1102097,
    "Region": "Zuid-Limburg",
    "Latitude": 50.96572,
    "Longitude": 5.60555,
    "Model": "WOFOST",
    "Simulation": "waterlimited",
    "Year": 2020,
    "Date": [ "2020-05-20", "2020-05-31", "..." , "2020-12-20", "2020-12-31" ],
    "TAGP_at_Date": [ 41.44378, 347.10842, "..." , 15950.85701, 15950.85701 ],
    "TWSO_at_Date": [ 0.0, 0.0, "..." , 3821.31834, 3821.31834 ]
}
```

Listing 2: A single sample JSON output record after aggregation

crop simulations. We have experimented with such configurations by using our university's HPC and the larger facilities from the Poznan Supercomputing and Networking Center (PSNC, http://www.psnc.pl). A supercomputer can provide many compute resources at relatively low cost, but they are very static and are usually operated in batch mode. Every processing job is entered into a queue and then has to wait its turn and until all requested resources are available. Running Spark on a Kubernetes (K8S) cluster provides a more dynamic approach but is typically also more expensive. The compute resources in a K8S cluster can be increased and decreased on demand, and Spark can use them directly to do additional work or more work at the same time. We have used Spark in a small K8S cluster provided by UBITECH (http://ubitech.eu) in various configurations to test processing scalability.

For deployment, the programming code to access input data, run crop simulations, and aggregate the large amount of simulation results needs to be combined into a Spark application. This needs to be packaged together with all dependencies (code libraries) it requires, so that Spark can distribute everything across all available worker nodes. A script, called `spark-submit`, is available to launch the application on a cluster. Depending on the `cluster manager` (e.g. SLURM or Kubernetes), this submit script can be called directly from the command line, or integrated into a cluster-specific deployment script. In the listing C1 (in Appendix C) an example is given for a small-scale deployment (using 4 nodes) in our university HPC. In this case, SLURM will assign the requested nodes, and Spark will then automatically take care of setting them up as *worker nodes*, distribute the software (and data if needed) and start the application as *driver program*. While Spark handles most of the needed distribution and collection of data over the nodes, the log messages that might get written on each node are typically handled and automatically transferred to the user's home directory on the cluster by the cluster management software.

2.7. Summary

For this study, we took the standard and open source Apache Spark big data analytics framework and tested if it could be used to run large numbers of crop simulations with an existing version of the WOFOST crop growth model. Although Spark provides API's for Python and R, it is JVM-based at its core so we chose to use the Java WISS-WOFOST implementation to avoid performance loss due to data marshaling between programming environments and runtimes. All input data needed for the simulations were extracted from a relational database, denormalized, and stored in MongoDB or MinIO. These NoSQL types of storage better support distributed and parallel data access and can avoid I/O bottlenecks. Regular Spark SQL commands were used to aggregate the results of all individual crop simulations to grid cell and regional levels. The setup has been tested and benchmarked on a small Kubernetes cluster and large HPC configurations, which is further described in the following section.

3. Results

3.1. Overall performance

To benchmark the Spark-based WISS-WOFOST approach, we analyzed the execution times of a test data set on different hardware configurations (see Section Experiments setup) and compared them with the execution time of the current WOFOST implementation used in MCYFS (the baseline). Table 1 summarizes the different hardware specifications used. *Parallelization* indicates the number of processing elements (see Section 2.6) used to perform the crop simulations (in parallel), while *Processing* lists the total execution time measured. Note that MCYFS did not fully use all available cores. The total workload (*Simulations* in the table) for the operational system by default consists of more than 3.7M simulations, while the test data set we used is much smaller, 113 K simulations. This has been taken into account when computing the execution time per simulation ($Time/Sim_{raw}$). Moreover, the results for the MCYFS system only contain the execution time for *Processing* while for the Spark-based system the results cover both *Processing* and *JSON preparation* taking into account that I/O reads have to be done in both cases. That is, MCYFS retrieves the simulation data directly from its database, while in the other cases this data is first extracted from the database into JSON files, which are subsequently loaded into the system. Both steps are part of the indicated *JSON preparation* time, which is constant here due to Spark only using the *driver program* node for the read I/O in these experiments.

The single-core normalized execution time ($Time/Sim_{scn}$) corrects for the number of processing elements that are used to execute the WOFOST simulations by inverse scaling (e.g. the total time would double when going from 2 cores to 1). Like the *raw* value it includes both the JSON preparation time when applicable and the crop simulation processing time. For simplicity, perfect parallelization is assumed in the calculation. In reality, this is never the case and actual *scn* values can be expected to be a little lower.

The results demonstrate that the Spark-based framework considerably reduces the processing time required for individual simulations compared to the baseline MCYFS system. Also, the use of more modern computer architectures still has a significant impact given that the Linux server with an older Intel Xeon E5 generation CPU is far slower (single-core normalized execution times) compared to the high-end Laptop (Intel Core i9) and the clusters 1–3. Moving from single

Table 1
Overview of various hardware configurations and measured crop simulations processing times. The $\texttt{Time/Sim}_{\texttt{raw}}$ times include both the processing time and the JSON preparation time when available. From that the $\texttt{Time/Sim}_{\texttt{scn}}$ single-core normalized execution time is calculated by inverse-scaling and assuming perfect parallelization.

Characteristic	MCYFS	Server	Laptop	Cluster 1	Cluster 2	Cluster 3
CPU type	Xeon E5	Xeon E5	Core i9	Xeon E3	Xeon E3	Xeon E3
Clock	2.3 GHz	3.2 GHz	2.4 GHz	2.4 GHz	2.4 GHz	2.4 GHz
Cores	20	12	16	1	2	6
Memory	128 GB	40 GB	16 GB	3 GB	3 GB	3 GB
Parallelization	10	12	16	1	2	6
JSON preparation	N/A	18.2 min	18.2 min	18.2 min	18.2 min	18.2 min
Processing	4153 min	14.7 min	2.4 min	34.0 min	20.0 min	7.5 min
Simulations	3.773.853	113.662	113.662	113.662	113.662	113.662
$\texttt{Time/Sim}_{\texttt{raw}}$	0.0660 s	0.0174 s	0.0109 s	0.0276 s	0.0202 s	0.0136 s
$\texttt{Time/Sim}_{\texttt{scn}}$	0.6600 s	0.2088 s	0.1744 s	0.0276 s	0.0404 s	0.0816 s

Table 2
Processing times for all maize and winter-wheat crop simulations between 2000–2020 from the MCYFS EU27 archive, on a Kubernetes cluster with various configurations.

Spark Executors	Total Cores	Processing time	
		Maize (3.8M sims)	Winter-wheat (5.1M sims)
1	4	670.9 min	820.9 min
2	8	324.9 min	445.9 min
4	16	164.1 min	219.2 min
8	32	84.3 min	137.8 min
16	64	48.4 min	88.2 min

Table 3
Processing times for 3.8M maize simulations with different numbers of partitions, measured on a Kubernetes cluster with various configurations.

Spark Executors	Total Cores	Processing time (Maize, 3.8M sims)		
		200 partitions	400 partitions	800 partitions
1	4	670.9 min	**668.9 min**	681.7 min
2	8	324.9 min	**322.9 min**	330.2 min
4	16	164.1 min	**162.9 min**	183.2 min
8	32	84.3 min	**83.1 min**	98.5 min
16	64	48.4 min	**47.6 min**	56.5 min

Table 4
Processing times for 5.1M winter-wheat simulations with different numbers of partitions, measured on a Kubernetes cluster with various configurations.

Spark Executors	Total Cores	Processing time (Winter-Wheat, 5.1M sims)		
		200 partitions	400 partitions	800 partitions
1	4	**820.9 min**	827.4 min	836.0 min
2	8	**445.9 min**	461.3 min	461.3 min
4	16	**219.2 min**	228.6 min	239.7 min
8	32	**137.8 min**	144.7 min	164.8 min
16	64	**88.2 min**	94.4 min	104.6 min

machines to multiple machines for the small clusters shows both the advantage of the overall processing times being significantly reduced due to the distribution of the workload, but also the cost of the increasing overhead by the higher *Time/Sim*$_{scn}$ values when more compute nodes are being introduced.

Some clarifying remarks apply to these benchmarks: (1) The measurements are only useful for *relative comparison* between the various configurations, the calculated single-core normalized execution times are indicative at best. (2) JRC is working on a new MCYFS system that is based on a similar distributed approach based on the Python implementation of WOFOST (PCSE), which is said to show comparable computational behavior as the WISS-WOFOST based implementation described here. (3) The MCYFS processing measurements in the table have been calculated by analyzing application log files of 10 parallel tasks that run crop simulations on the system, instead of active monitoring of a running system (which was no longer available at the time of writing). However, this should not affect the results. (4) We did not compensate for the higher clock rate of the (Linux) server (3.2 GHz versus 2.3–2.4 GHz of the other configurations), since the net effects are hard to estimate and can only increase the gap in performance already shown.

3.2. System scaling

In addition to comparing performance, we also analyzed the scalability of the system. Specifically, we looked at how adding more compute resources affected the total runtime required for processing two larger crop simulation input data sets. For this study, we selected all maize and winter wheat simulations between 2000 and 2020 from the MCYFS EU27 archive. The first data set has 3.8M (3 869 586) records and the second 5.1M (5 137 804) (each record contains all the data needed for a single-crop growth simulation). Both were used as input for running crop simulations on a Kubernetes cluster consisting of 1, 2, 4, 8, and 16 nodes (and the same number of Spark Executors) with 4 cores and 16 GiB per worker node. Table 2 shows the total processing times measured per experiment.

The plot of the processing times on a graph (see Fig. 4) clearly shows the relation with the number of worker nodes used and that after a certain number of nodes adding more resources will no longer be cost-effective.

The general scaling behavior is similar for the 3.8M (maize) and 5.1M (winter-wheat) simulations. However, an important difference between these two data sets is that the latter, winter wheat, is a winter crop that has a significantly longer growing season (the period between sowing and harvest). So not only does the data set have more simulation records, each record also requires more storage space due to the longer time series of weather data it contains (the weather data are the largest component of a simulation record). Since Spark splits the total workload (all simulations to perform) into a specified number of partitions to be processed in parallel (using the cores available to each Spark Executor), both mentioned factors affect the memory space needed per partition, as well as the JVM GC (garbage collection) (background) process that periodically frees up deallocated memory blocks. Besides that, depending on the processing workflow partitions sometimes have to be exchanged between the worker nodes (an operation known as a shuffle in Spark), which is a time consuming operation. These, and other considerations, make the number of partitions an important tuning parameter for Spark performance. Tables 3 and 4 illustrate this. On the same system, using 400 partitions is more performant for the maize simulations, while for winter-wheat it is 200 partitions (more optimal choices might exist). In these two cases, the effect is not very large. However, in an operational setting, such differences add up over time and are worth keeping in mind.

The second study of system scalability was performed using the HPC of the PSNC Eagle cluster. This has a large number of nodes with 2×14 cores and 64-128-256 GiB memory, from which we used 32, 48 and 64

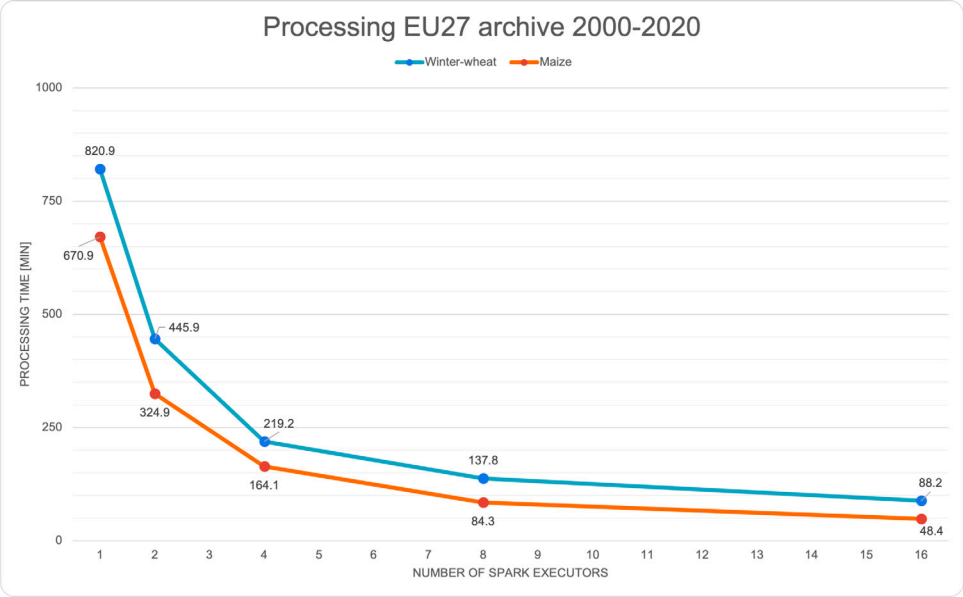

Fig. 4. Plot of the processing times for all the maize and winter-wheat crop simulations between 2000–2020 from the MCYFS EU27 archive, on a Kubernetes cluster with various configurations. The diminishing benefit of adding more nodes is clearly visible.

Table 5
Maize and winter-wheat crop simulations processing times on a HPC cluster using various configurations with large numbers of cores.

Spark Executors	Total Cores	Processing time	
		Maize (3.8M sims)	Winter-wheat (5.1M sims)
32	224	5.43 min	13.81 min
48	336	5.20 min	10.25 min
64	448	4.79 min	7.55 min

instances (with one Spark Executor each) and assigned with 7 cores and 8 GiB memory. In practice, the cluster manager could therefore combine two of these instances on a physical node or keep them separated; however, this should not affect the performance measurements. There is a clear, and to be expected, significant advantage of the HPC hardware (such as Infiniband networking with very high throughput and very low latency) over the Kubernetes Cloud hardware. Even when considering the increase in Spark Executors and the number of cores assigned per Executor. The measured execution times for the crop simulations are summarized in Table 5. The processing of the larger winter-wheat data set also allows Spark to use additional worker nodes and cores more than the smaller maize data set, where the advantages are marginal. However, comparing the fastest processing of all maize simulations with the standard MCYFS system, we observe a significant 99% reduction in total runtime on hardware that has a 64-times higher distribution factor (64 nodes versus 1 node) and in total uses roughly 45 times more cores (448 cores versus 10 used by MCYFS).

3.3. Data aggregation

As discussed in Section 2.6.3 the total workload usually consists of two parts. The execution of a large number of crop simulations for grid cell/SMU / STU combinations, followed by aggregation of the simulation results first to grid cells and later to administrative regions for reporting. In the MCYFS an Oracle relational database is used to calculate the aggregated data, after first all simulation results have been ingested (and indexed). This loading of the data can take some time, but after that the needed database operations are typically fast. As an alternative option, we explored using Spark for the initial data aggregation (to grid cells). Due to the partitioning of the data on networked nodes, the required `groupBy` and similar Spark operations are known to be costly (in time). However, aggregating with Spark significantly reduces the amount of data needing to be imported into the database afterward, and it might not always be required to store the full detailed output of every crop growth simulation run (when necessary, they are fast to rerun).

Table 6 shows the measured processing times of the two data sets in the Kubernetes system. For this experiment, we only used the configurations with 8 and 16 nodes, again with 1 Spark Executor per node and 4 cores assigned to each executor. For simplicity, we estimate the time needed for the data aggregation by comparing the processing time of all simulations followed by the aggregation and the processing time of all simulations only. Regular SQL operations, such as used for the aggregation, are heavily optimized by Spark, and combined with the required data shuffling, a detailed analysis of the system in operation would be needed to get more precise numbers. We considered this unnecessary for the more superficial comparisons made here.

Comparing aggregation and data loading times between the MCYFS and Spark-based approaches is also complicated due to all the differences between the systems. However, to get at least an impression, we took the following approach. For both MCYFS (see Table 1 for the configuration) and Kubernetes deployment with 16 Spark Executors (using 64 cores total), we collected the processing times for crop simulations, data aggregation, and database loading, either by analysis of the application log files (in case of MCYFS) or by running the system (in case of Spark). To compare the aggregation performance in the database (using PL/SQL) and by Spark (Spark SQL), we estimated for both the number of STU records/sec and used this factor to calculate the estimated aggregation times for the two test data sets. Similarly, for database loading, we calculated the rows/sec loading speed and applied this factor to estimate the time it would take to load the Spark results into the Oracle database. The results are included in Table 7.

Table 6
Results for the maize and winter-wheat crop simulations followed by output aggregation from grid cell to regional level, on a Kubernetes cluster using 32 and 64 cores. Time used for the aggregation is estimated based on the measured total processing time.

Data set	Result	Configuration Spark Executors (cores used)	
		8 (32)	16 (64)
Maize (3.8M sims)	Overall processing time	89.24 min	50.44 min
	Only simulations	84.28 min	48.42 min
	Estimated aggregation time	4.96 min	2.02 min
Winter-wheat (5.1M sims)	Overall processing time	154.31 min	103.42 min
	Only simulations	137.82 min	88.16 min
	Estimated aggregation time	16.49 min	15.26 min

Table 7
Data aggregation and database loading times comparison between using the database for the aggregation (as done in MCYFS), and using Spark SQL to aggregate before inserting the results into a database (as done in our prototype system).

	MCYFS (agg in DB)		Spark (agg before DB)	
	Maize (3.8M)	Winter-wheat (5.1M)	Maize (3.8M)	Winter-wheat (5.1M)
DB Load	14.90 min	49.78 min	1.03 min	1.69 min
Data Agg	9.59 min	14.09 min	2.02 min	15.26 min
Total	24.49 min	63.87 min	3.05 min	16.96 min

Table 8
Results for the maize and winter-wheat crop simulations followed by output aggregation on a HPC cluster using 224, 336 and 448 cores. Time used for the aggregation is estimated based on the measured total processing time.

Data set	Result	Configuration Spark Executors (cores used)		
		32 (224)	48 (336)	64 (448)
Maize (3.8M sims)	Overall processing time	7.32 min	6.11 min	6.02 min
	Only simulations	5.43 min	5.20 min	4.79 min
	Estimated aggregation time	1.89 min	0.91 min	1.23 min
Winter-wheat (5.1M sims)	Overall processing time	18.13 min	15.78 min	10.89 min
	Only simulations	13.81 min	10.25 min	7.55 min
	Estimated aggregation time	4.32 min	5.53 min	3.34 min

We would like to note that with Spark, the data is aggregated before loading it into the database, while the MCYFS first loads all simulation results into the database and then performs the aggregation. In addition, the Spark configuration uses a significantly higher number of cores (64 versus 10 for the MCYFS), which helped to keep the aggregation time similar to the more optimized performance from the Oracle database. However, it clearly indicates that multiple aspects of a distributed system must be taken into account when evaluating its performance, as illustrated by Fig. 5 (based on Table 7).

As a final experiment, we used the PSNC HPC to run both crop growth simulations and data aggregation to the grid cell level with Spark SQL on large sets of nodes. For the results, see Table 8 and Fig. 6. On the maize data set the benefits of adding more nodes are small, and the estimated aggregation time even gets slightly worse when the data is distributed across more Spark Executors. Still, comparing the estimated aggregation time on 64 nodes with 448 cores total, with that for the standard MCYFS system (1 node with 10 cores used) shows that a 95% reduction in processing time is achieved. Interestingly, on the larger winter-wheat data set, the medium configuration (on 48 nodes) shows less good performance improvements, most likely because the number of partitions is not ideal for the number of Spark Executors available. This illustrates once more the importance of performance tuning of Spark based systems in order to get the best results from them.

3.4. Aggregation results

The aggregation process combines the output of the crop simulations first at grid cell level (25 × 25 km) and then further at the lowest regional level. See Section 2.6.3 for more details. Taking the simulations of the maize crops as an example, the process results in 6.555 values for the Total Above Ground Production (TAGP) in dry weight kg/ha of biomass and for the Total Weight of Storage Organs (TWSO) in dry weight kg/ha, in 2020 for regions in EU27. The maps of these results are shown in Figs. D.1 and D.2, in Appendix D. Further aggregation can be performed for larger administrative regions.

4. Discussion

Using a typical numeric crop growth simulation model and a widely used standard framework for big data processing and analytics on distributed systems, we successfully built a prototype of the core of a yield forecasting and monitoring application. We then benchmarked its performance and scalability characteristics on a few different hardware configurations based on ephemeral resources such as those that can be used from cloud or HPC providers. The measurements were based on the processing of existing data sets for the maize and winter wheat crops and for a climate scenario. Although the data required for a single simulation is not that extensive, the large total number of simulations to be processed and the amount of output data produced place the task in the big data domain.

The obtained results illustrate that the expected effect of scaling on total run-time for performing crop growth simulations with the process model also applies, and thus allows trading-off compute costs with required result availability (within certain bounds). *Based on these findings we conclude that currently available standard distributed computing frameworks, specifically Apache Spark, are sufficient for building a system*

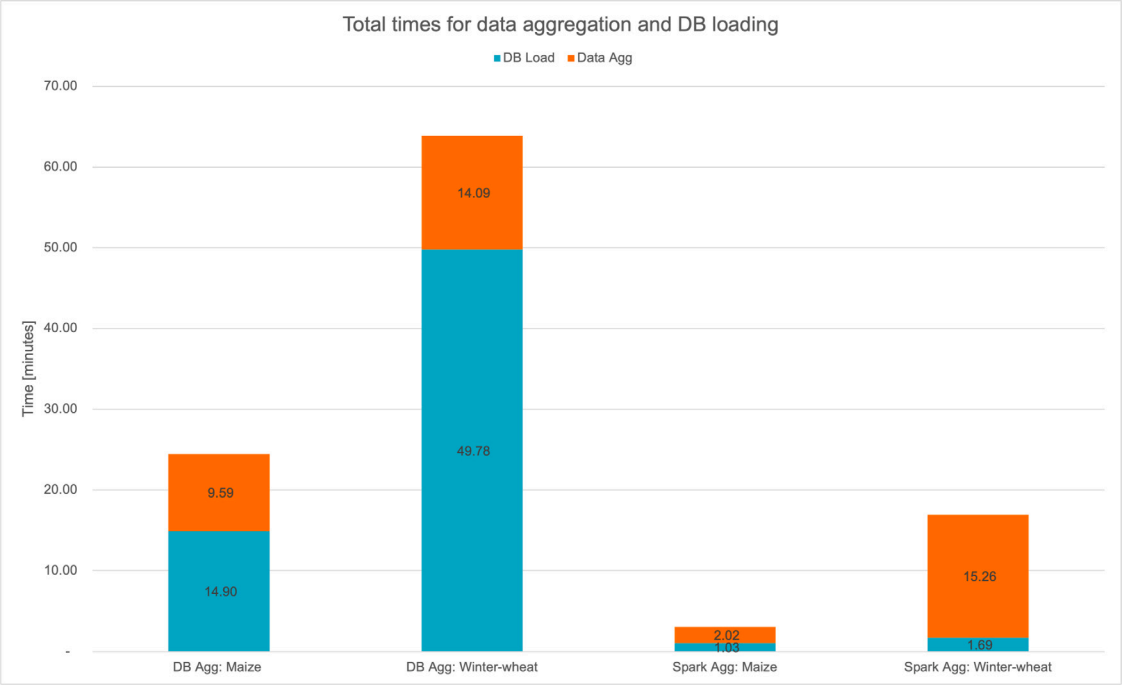

Fig. 5. Combined times for data aggregation and database loading, when using the database to do the aggregation and when using Spark SQL. Results shown for both the maize and winter-wheat crop simulations.

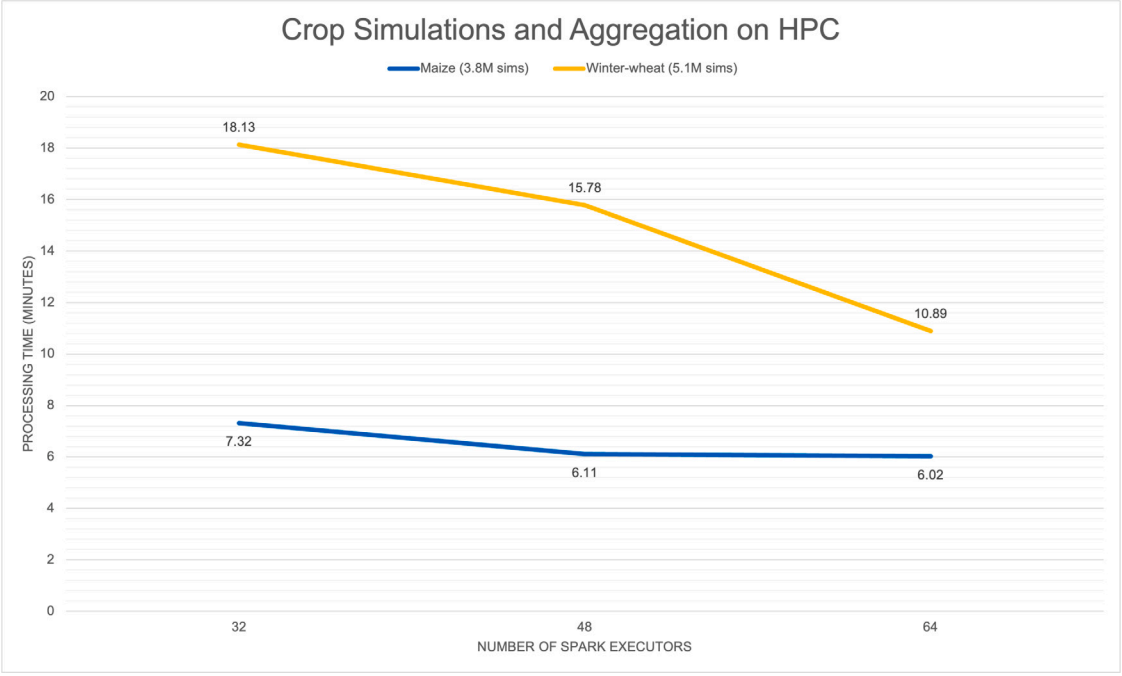

Fig. 6. Plot of the results for the maize and winter-wheat crop simulations followed by output aggregation on a HPC cluster using 224, 336 and 448 cores.

that is capable of running very large volumes of crop growth simulations. Furthermore, the following insights can be deduced:

- It is quite possible to base such a system on standard ephemeral compute resources managed by a commonly available open source framework, that hides a lot of the details of distributed computing, such as workload partitioning and hardware failure recovery, and makes it more easily accessible.
- A distributed system as described is capable of handling large amounts of crop simulations in reasonable time-frames, including all related (big) data management. The performed benchmarks provide insights into the performance and scalability that can be expected. As already common practice in other science domains such as climate sciences and hydrology, distributed technologies such as described in this paper are ready to be leveraged in the agricultural domain.
- Existing numerical models and data storage might need some adaptation before they can be *efficiently* used for distributed computing. In the case study little effort is needed to split a total workload into separate jobs that execute crop simulations simultaneously (in parallel computing this is referred to as an embarrassingly parallel workload). While an easy approach when applicable, it is not necessarily the most performant or power-efficient one. In this study little to no time has been spend on tuning the system to get the best possible performance. Spark has many configuration options that can be experimented with, the workload/simulations can be created or executed in a more efficient way, and the model (or parts of it) perhaps can be parallelized or replaced by faster solutions.
- A certain level of knowledge about the specifics of distributed computing and distributed data storage technologies will still be required, and comes with a learning curve that might not suite everyone. Specifically this will be needed when it comes to resolving issues (debugging the system) and for performance tuning.
- In the case study we configure a prototype system for distributed processing of crop growth simulations using a specific implementation (the Java version) of WOFOST, and set up an *extract–transform–load* (ETL) pipeline to ingest the input data it needs for the crop simulations. Both on top of the generic Spark framework. Due the flexibility of the framework for communicating with models written in JVM languages, Python, and R, as well as to any command line oriented model (compiled or interpreted) via standard Linux pipes, it should be straightforward to incorporate any of the other WOFOST implementations available (de Wit et al., 2019). Other kinds of crop models will require additional configuration since the APIs and data requirements of these models can vary (Porter et al., 2014; Janssen et al., 2017). Still, it can be done as well, using the same framework and the methodology described in this paper.
- The prototype provides only the core of a potential operational system, as noted in 2.1, this was sufficient for the aim of this paper. As a follow-up the work can be extended into a production oriented application with e.g., a user interface that can hide some or most of the complexity from domain experts for day-to-day use.
- Spark is a generic distributed framework for batch processing and analysis of big data. Instead of the numerical model used in the prototype other kinds of models, including machine learning model training and inferences, can equally well be used. Alternatively, there are other frameworks available that specifically focus on machine learning (particularly useful for deep learning), or streaming data processing. Fortunately the basic concepts of distributing workloads across computers are the same for all these frameworks and experience with one easily applies to others.
- Various kinds of simulation output data post-processing and analytics can already be done on the same distributed system, which can reduce the size of the final results, making them easier manageable. An advantage is that the numerical model used is deterministic. Since individual simulations can easily and quickly be reproduced, there might be no need to store all output data from every simulation performed.
- The scalability of the system follows the expected curve with high gains on the initial increases of compute nodes, and decreasing benefits when the system gets bigger. Within limits, it allows a trade-off between acceptable compute costs and overall calculation time to obtain the result. Such flexibility is a benefit, but it also requires changes in thinking about costs of computations. E.g. they might no longer be fixed department IT costs, but rather become related to specific tasks or projects.

Although distributed technologies proved to be suitable for the case study and similarly can be expected to be equally applicable to other big data processing and analytics that are or will be part of digital agriculture, e.g. for the processing of streams of data from many IoT devices (sensors) or for the computationally intensive 4D-Var data assimilation (Huang et al., 2019), they are yet not commonly known in the agricultural informatics domain, where most agricultural modelers and data scientists will first have to invest in the additional knowledge required. Naturally, the kind and depth of required expertise will vary, ranging from basic awareness of the existence and possibilities of the technologies to expert and practical knowledge of distributed computing frameworks and e.g. functional programming (as mentioned in Section 2.4).

It can also be an important step going from personally curated data sets stored and processed on a local server to working with distributed, remote (perhaps even Cloud based), storage and processing of data. Some of the changes required are described in Section 2.5. However, in addition to those technology-oriented considerations, it can also impact broader organizational aspects since distributed computing and the sharing of resources such as infrastructure and data can go hand in hand. Furthermore, as a related aspect, users, managers, and developers will be confronted with new and different cost models, e.g., pay per use of temporarily allocated compute resources, which might not readily fit into the classical budgeting of IT resources.

Although the implemented prototype system described in this paper covers everything needed to create an operational version of it, there certainly remain a few interesting topics for future work:

(i) In our approach, we take advantage of the WISS-WOFOST model implemented in Java, which matches well with Apache Spark, which is JVM-based. However, Apache Spark also supports Python, so a similar study could be performed using PySpark and the PCSE implementation of WOFOST (written in Python). This might, however, incur a performance cost since data will need to be marshaled between the Python and the JVM environments.

(ii) The described implementation is rather straightforward and takes an embarrassingly parallel approach for the computations. A possible optimization is the minimizing of the number of crop growth simulations (by sorting or caching results) that actually needs to be performed, e.g. by smart use of the named parameters we already included in the data scheme.

(iii) As noted above, the weather data make up the largest part of each simulation input record. A possible improvement then is the broadcasting of these data to worker nodes and constraining crop simulation runs to those nodes that have the required weather data. This would reduce the simulation record size, overall IO need, and thus increase performance/simulation throughput.

(iv) Spark uses an optimization technique called adoptive query execution, which uses run-time statistics to select the most efficient query execution plan. This works well for Spark SQL; however, the crop growth simulation process model is an opaque data transformation step

in the processing pipeline to Spark, and therefore it cannot include it in its optimizations. If this can be improved, it should be beneficial for the overall performance.

(v) The data currently used includes (text) IDs for grid cells and administrative regions, which allow for simple text-based aggregations. Future data sets and aggregation approaches might not have this convenience and thus need support for geospatial data types and operations. There is quite a bit of history and development of spatial extensions on top of the Spark framework (both JVM and Python based), e.g. Apache Sedona (https://sedona.apache.org/) where many initiatives seem to end up. Integrating it into the system described in this paper would give it real large-scale spatial data processing and analysis capabilities.

(vi) Although in this work a traditional numerical crop growth simulation model is used, deep learning and machine learning are increasingly prominent in agricultural data analysis and modeling, especially with large datasets. In addition to the comparison already suggested with the Python PCSE implementation of WOFOST, evaluations can be performed with machine learning-based models as the baseline, resulting in a broad overview of current technologies for high-efficiency agricultural modeling.

Note that while standard frameworks and cloud or HPC resources certainly make distributed computing more accessible and easier to use, building such an operational system is still not trivial. In particular, when it is the first encounter with building a distributed application, or with deploying such application on distributed cloud or HPC hardware. A framework such as Apache Spark can handle and hide many of the low-level details (data partitioning, task scheduling, handling batch and streaming data, hardware failure recovery, etc.), besides providing extensive data science and machine learning libraries for data analytics. However, ultimately it is still beneficial to know how the system is working and how it can be tuned to achieve the best possible performance, with acceptable costs. Fortunately, such knowledge is domain-agnostic and applicable to distributed systems in general, and thus can be left to specialized data engineers.

5. Conclusions

Considering the research objective, the case study described in this paper and the benchmark results show that *a usable system can indeed be constructed on top of standard technology for big data analytics and distributed computing, in this case Apache Spark, and that it can provide a solution when true big data processing and analytics are required in the agricultural domain*. Generally speaking, the chosen case study is a regular big data processing task, slightly complicated by the use of the numerical crop simulation model, which is not regular SQL that the framework knows how to fully optimize, but will remain a black box that can only be handled in a generic way. Still, the prototype leaves room for further optimization and fine-tuning of the system. Furthermore, the applicability of the approach is not limited to this use case with a crop growth model. Due to the flexibility of the Spark framework for embedding e.g. Python and command-line based (compiled) models as well, it is a very generic solution.

In case the data volume or computational requirements justify it, distributed computing using these standard technologies is a viable and powerful approach in digital agriculture. With the expected increase in demand for scalable processing — due the ongoing transformations toward data-driven agriculture — these technologies will become increasingly relevant. However, their use should be dictated by actual needs, due to the additional complexity involved.

It is important for stakeholders in agriculture — agronomists, data scientists, and (spatial) data engineers — to understand distributed technologies, each in line with their role and expertise. Sensible application and cross-disciplinary collaboration will be key to unlocking the full potential of distributed computing in agricultural informatics.

CRediT authorship contribution statement

Rob Knapen: Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Allard de Wit:** Writing – original draft, Visualization, Methodology, Formal analysis, Data curation, Conceptualization. **Eliya Buyukkaya:** Writing – original draft, Software, Data curation. **Petros Petrou:** Software, Resources, Methodology, Investigation. **Dilli Paudel:** Writing – review & editing, Validation. **Sander Janssen:** Writing – review & editing, Supervision, Funding acquisition, Conceptualization. **Ioannis Athanasiadis:** Writing – review & editing, Supervision, Funding acquisition, Conceptualization.

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.

Acknowledgments

The work described in this document has been carried out as part of the CYBELE research project, which has been funded by the European Commission under the Horizon 2020 Program (Grant Agreement No. 825355).

Appendix A

See Tables A.1–A.7.

Table A.1
Overview of the schema for the crop simulation *input data*. Further tables show the details of subsections.

```
root
 |-- version: long
 |-- simId: string
 |-- simModel: string
 |-- simType: string
 |-- simCropId: long
 |-- simCropName: string
 |-- simCropVariety: long
 |-- simYear: long
 |-- simStartDate: string
 |-- simEndDate: string
 |-- location: struct
 |    |-- type: string
 |    |-- coordinates: array
 |    |    |-- element: double
 |-- sourceType: string
 |-- sourceDetails: struct
 |    |-- name: string
 |    |-- grid: long
 |    |-- smu: long
 |    |-- stu: long
 |    |-- altitudeM: long
 |    |-- gridWeightFactor: double
 |-- cropParams: struct
 |-- soilParams: struct
 |-- siteParams: struct
 |-- agromanagement: struct
 |-- meteo: struct
```

Table A.2
Sub-schema for the meteo input data for a simulation. For efficiency the variables are stored as arrays which need to match the date range.

```
|-- meteo: struct
|    |-- version: long
|    |-- id: string
|    |-- startDate: string
|    |-- endDate: string
|    |-- details: struct
|    |    |-- grid: long
|    |-- data: struct
|    |    |-- temperatureMin: array
|    |    |    |-- element: double
|    |    |-- temperatureMax: array
|    |    |    |-- element: double
|    |    |-- temperatureAvg: array
|    |    |    |-- element: double
|    |    |-- vapourPressure: array
|    |    |    |-- element: double
|    |    |-- windSpeed10M: array
|    |    |    |-- element: double
|    |    |-- windSpeed2M: array
|    |    |    |-- element: double
|    |    |-- precipitation: array
|    |    |    |-- element: double
|    |    |-- e0: array
|    |    |    |-- element: double
|    |    |-- es0: array
|    |    |    |-- element: double
|    |    |-- et0: array
|    |    |    |-- element: double
|    |    |-- radiation: array
|    |    |    |-- element: double
```

Table A.3
Sub-schema used for all crop, soil, site and agro-management parameters that are relevant but only need to be stored and passed to the model to run a simulation. For flexibility text string representations are used here.

```
|-- parameters: struct
|    |-- id: string
|    |-- params: array
|    |    |-- element: struct
|    |    |    |-- name: string
|    |    |    |-- unit: array
|    |    |    |    |-- element: string
|    |    |    |-- value: array
|    |    |    |    |-- element: string
```

Table A.4
Overview of the schema for the crop simulation *output data*. Further tables show the details of the subsections. The `message` field is used to save any error messages that occur during the simulation, for later inspection.

```
root
|-- description: struct
|-- message: array
|    |-- element: string
|-- timeseries: struct
|-- summary: struct
```

Table A.5
Sub-schema for the simulation output summary, consisting of the main WOFOST output variables, such as the Total Above Ground Production (`TAGP`) and the Total Weight of the Storage Organs (`TWSO`).

```
|-- summary: struct
|    |-- dvs: double
|    |-- laimax: double
|    |-- tagp: double
|    |-- twso: double
|    |-- ctrat: double
|    |-- cevst: double
|    |-- rd: double
|    |-- dos: long
|    |-- doe: long
|    |-- doa: long
|    |-- dom: long
|    |-- doh: long
|    |-- dov: long
```

Table A.6
Sub-schema for the descriptive information about the crop simulation, indicating amongst others the location, the type of crop, and which crop growth simulation has been run.

```
|-- description: struct
|    |-- simId: string
|    |-- simModel: string
|    |-- simType: string
|    |-- simCrop: long
|    |-- simCropName: string
|    |-- simCropVariety: long
|    |-- simYear: long
|    |-- simStartDate: string
|    |-- simEndDate: string
|    |-- location: struct
|    |    |-- coordinates: array
|    |    |    |-- element: double
|    |    |-- type: string
|    |-- sourceType: string
|    |-- sourceDetails: struct
|    |    |-- altitudeM: long
|    |    |-- grid: long
|    |    |-- gridWeightFactor: double
|    |    |-- name: string
|    |    |-- smu: long
|    |    |-- stu: long
```

Table A.7
Sub-schema with the time series data of key WOFOST variables during the simulation. These can be helpful to solve simulation issues.

```
|-- timeseries: struct
|    |-- date: array
|    |    |-- element: string
|    |-- elapsed: array
|    |    |-- element: integer
|    |-- dvs: array
|    |    |-- element: double
|    |-- lai: array
|    |    |-- element: double
|    |-- tagp: array
|    |    |-- element: double
|    |-- twso: array
|    |    |-- element: double
|    |-- ctrat: array
|    |    |-- element: double
|    |-- rd: array
|    |    |-- element: double
|    |-- sm: array
|    |    |-- element: double
|    |-- wwlow: array
|    |    |-- element: double
```

Appendix B

```scala
import nl.wur.json.{SimulationInput, SimulationOutput}
import nl.wur.wiss.core.{SimXChange, TimeDriver}
import nl.wur.wissmodels.wofost.WofostModel

object WofstRunner extends Serializable {
    import scala.collection.JavaConverters._

    def run(input: SimulationInput): Try[SimulationOutput] = Try {
        // get the input parameters and prepare an output instance
        val output = new SimXChange(input.getSimId)
        val result = new SimulationOutput()

        // fill a ParXChange instance from the input data
        val params = input.getParXChange

        // perform a daily timestep based simulation
        new TimeDriver(new WofostModel(params, output)).run()

        // collect input and output details into the result
        result.updateDescription(input, output)
        result.updateTimeSeriesSummary(input, params, output, true)
        result
    }
}
```

Listing B1: Code for embedding the WISS-WOFOST model in a Scala function

```scala
val spark = SparkSession.builder()
    .appName("WOFOST-Simulations")
    .getOrCreate()

// set up the encoders for the Dataset (row) types
implicit val inputEncoder: Encoder[SimulationInput] =
    Encoders.bean(classOf[SimulationInput])
implicit val outputEncoder: Encoder[SimulationOutput] =
    Encoders.bean(classOf[Simulationoutput])

// define the (JSON) input dataset
val inputData : Dataset[SimulationInput] = spark.read
```

```
        .option("encoding", "UTF-8")
        .format("json")
        .load(/* inputDataPath */)
        .as[SimulationInput]

    // method to run simulations for all input records
    def runByMapPartition(nPartitions): Dataset[SimulationOutput] = {
        inputData
            .repartition(nPartitions)
            .mapPartitions(_.map(WofostRunner.run))
    }

    // define how to produce wofost simulation results
    val outputData = runByMapPartitions(/* partition count */)
        .cache()

    // extract all successful simulations and save them
    outputData
        .filter(array_contains(col("message"), "Ok"))
        .write
        .format("json")
        .mode(SaveMode.Overwrite)
        .save(/* failedSimulationsPath */)
```

Listing B2: Example of running WISS-WOFOST with Spark

```
    // define how to produce wofost simulation output data
    val outputData = runByMapPartitions().cache()

    // define how to produce the aggregated result
    val aggData = outputData
      // add a column with 0 if simulation succeeded and 1 if there were errors
      .withColumn("err", (!array_contains(col("message"), "Ok")).cast("integer"))
      .select(
        col("err"),
        col("description.simCrop").as("crop_code"), col("description.simCropName").as("crop_name"),
        col("description.simCropVariety").as("crop_variety"), col("description.simYear").as("year"),
        col("description.sourceDetails.grid").as("grid"), col("description.sourceDetails.name").as("name"),
        col("description.location.coordinates").as("coordinates"),
        col("description.simModel").as("sim_model"), col("description.simType").as("sim_type"),
        col("description.sourceDetails.gridWeightFactor").as("weight_factor"),
        // expression so that the sum of weight factors for successful simulations can be calculated
        expr("""case when err = 0 then description.sourceDetails.gridWeightFactor else 0.0 end""").as("non_err_weight"),
        // the timeseries are arrays, they require a bit more complex processing
        col("timeseries.elapsed").as("day"), col("timeseries.date").as("date"),
        expr("""transform(timeseries.tagp, x -> x * description.sourceDetails.gridWeightFactor)""").as("w_tagp"),
        expr("""transform(timeseries.twso, x -> x * description.sourceDetails.gridWeightFactor)""").as("w_twso")
      )
      .groupBy(
        col("crop_code"), col("crop_name"), col("crop_variety"), col("year"), col("grid"), col("name"),
        col("coordinates"), col("sim_model"), col("sim_type"), col("date"), col("day")
      )
      .agg(
        count("*").as("n_sims"),
        sum("err").as("sum_err"), // sum the number of simulations that had errors
        sum("weight_factor").as("sum_weights"), // sum all weight factors
        sum("non_err_weight").as("sum_non_err_weights"),
        // element wise summing of the values in the arrays
        elementwiseSumDoubleArrays(collect_list("w_tagp")).as("sum_w_tagp"),
        elementwiseSumDoubleArrays(collect_list("w_twso")).as("sum_w_twso")
      )
      .select(
        col("sum_err").as("ERR"), expr("round(sum_weights, 5)").as("Total_Weights"),
        expr("round(sum_non_err_weights, 5)").as("NonErr_Weights"),
```

```
        col("crop_code").as("Crop_Code"), col("crop_name").as("Crop_Name"),
        col("crop_variety").as("Crop_Variety"),
        col("grid").as("Grid"), col("name").as("Region"),
        col("coordinates").getItem(1).as("Latitude"), col("coordinates").getItem(0).as("Longitude
    "),
        col("sim_model").as("Model"), col("sim_type").as("Simulation"),
        col("year").as("Year"), col("date").as("Date"),
        // element wise processing again of the arrays
        expr("""transform(sum_w_tagp, x -> round(x / sum_weights, 5))""").as("TAGP_at_Date"),
        expr("""transform(sum_w_twso, x -> round(x / sum_weights, 5))""").as("TWSO_at_Date")
      )
      .orderBy(col("Crop_Code"), col("Grid"), col("Year"), col("Day"))

    // terminal action to perform the aggregation and save the result
    aggData.write.format(outputFormat).mode(SaveMode.Overwrite).save(savePath)
```

Listing B3: Code for the data aggregation with Spark

Appendix C

```
# -= slurm Anunna HPC =-
#SBATCH --time=02:00:00
#SBATCH --mem-per-cpu=4G
#SBATCH --nodes=4
#SBATCH --cpus-per-task=2
#SBATCH --job-name="wofost-runs"
#SBATCH --qos=std

module load spark/3.2.3-2.7
source $SPARK_HOME/wur/start-spark

# Spark config
spark_deploy_mode=client
spark_master=local[*]
spark_driver_memory=2g
spark_executor_memory=1g

# Set to proper path and file (needs to be absolute)
log4jconf=file:///home/WUR/[user]/wofost/wofost-spark-submit-v1/log4j.properties
# Folder with jars (can be relative)
jarsdir=./jars

# submit the spark job
spark-submit \
  --master ${spark_master} \
  --deploy-mode ${spark_deploy_mode} \
  --driver-memory ${spark_driver_memory} \
  --executor-memory ${spark_executor_memory} \
  --class nl.wur.json.WofostRun \
  --jars "${jarsdir}/WISSFramework-1.0.jar,${jarsdir}/WOFOST-WISS-7.2.jar,${jarsdir}/jafama.jar"
    \
  --conf "spark.eventLog.enabled=false" \
  --conf "spark.executor.extraJavaOptions=--XX:+PrintGCDetails --XX:+PrintGCTimeStamps" \
  --conf "spark.executor.extraJavaOptions=-Dlog4j.configuration=${log4jconf}" \
  --driver-java-options "-Dlog4j.configuration=${log4jconf}" \
  --files /lustre/scratch/WUR/[user]/wofost_inputs/[crop-simulation-input-data].json \
  ${jarsdir}/WOFOST-JSON-1.0-jar-with-dependencies.jar --url ${spark_master} \
  --mode json \
  --repartition 8 \
  --in [crop-simulation-input-data].json
```

Listing C1: Deploying a Spark application with spark-submit and SLURM

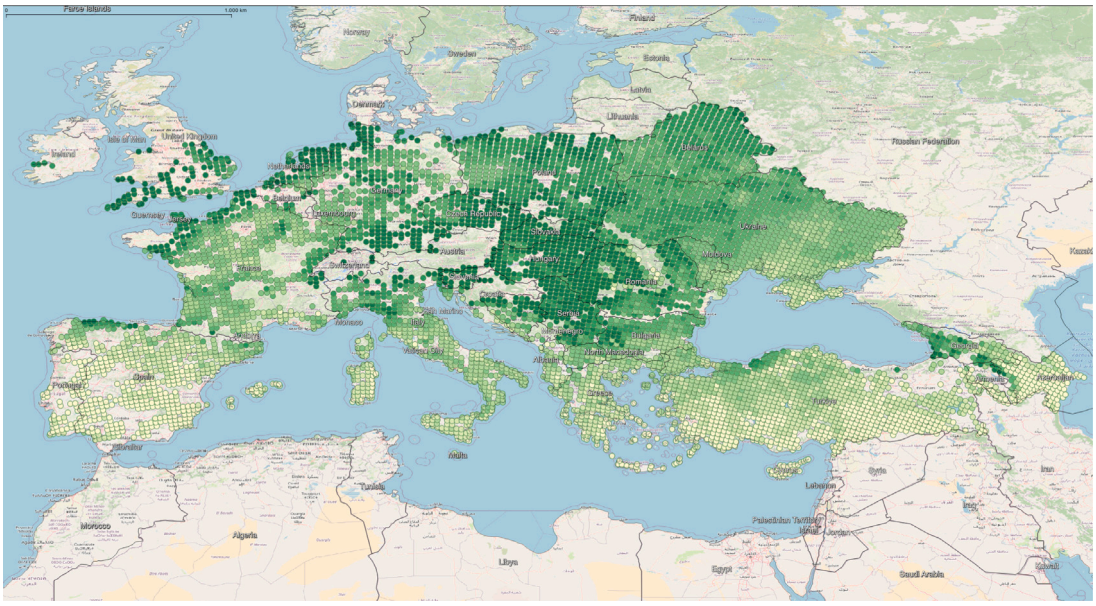

Fig. D.1. Map of aggregated Total Above Ground Production (TAGP) as dry weight in kg/ha from the 3.8M Maize crop simulations (2020, WOFOST water-limited). Showing lowest (700 kg/ha) to highest (29.000 kg/ha) values in light to dark shades of green.

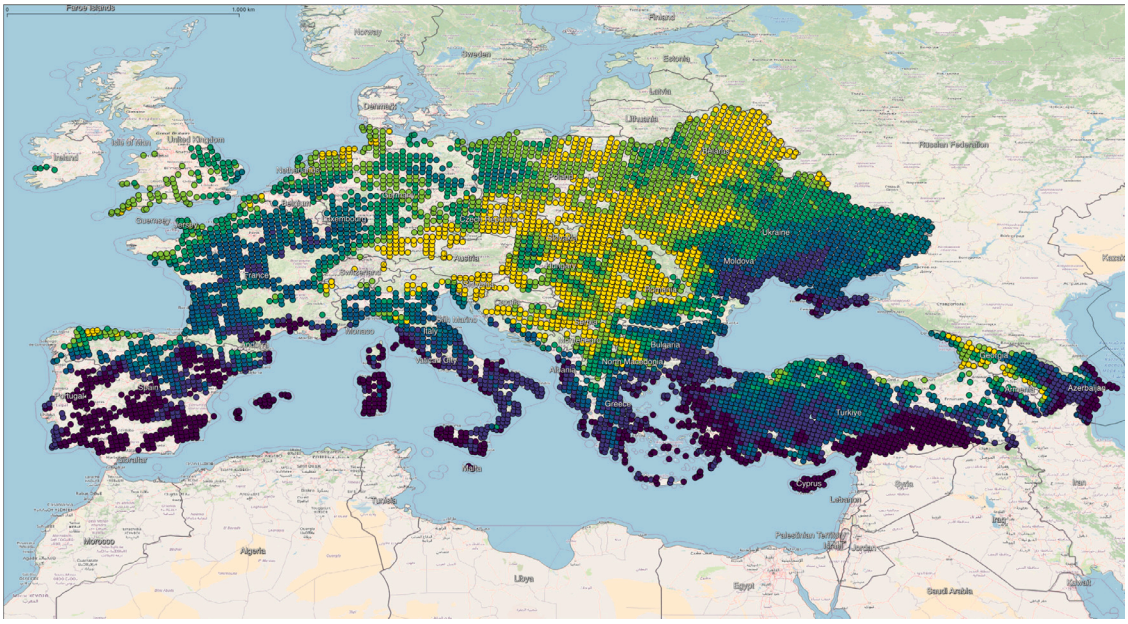

Fig. D.2. Map of aggregated Total Weight of Storage Organs (TWSO) as dry weight in kg/ha from the 3.8M Maize crop simulations (2020, WOFOST water-limited). Showing lowest (0 kg/ha) values in purple, via blue and green, to highest (14.600 kg/ha) values in yellow.

Appendix D

See Figs. D.1 and D.2.

Data availability

Data will be made available on request.

References

Afshar, M.H., Foster, T., Higginbottom, T.P., Parkes, B., Hufkens, K., Mansabdar, S., Ceballos, F., Kramer, B., 2021. Improving the performance of index insurance using crop models and phenological monitoring. Remote. Sens. 13 (5), http://dx.doi.org/10.3390/rs13050924.

Alderman, P., 2021. Parallel Gridded Simulation Framework for DSSAT-CSM (version 4.7.5.21) Using MPI and NetCDF. Copernicus GmbH, http://dx.doi.org/10.5194/gmd-2021-183.

Backus, J., 1978. Can programming be liberated from the von Neumann style? Commun. ACM 21 (8), 613–641.

Boogaard, H., van der Grijn, G., Agrometeorological Indicators from 1979 to Present Derived from Reanalysis. http://dx.doi.org/10.24381/cds.6c68c9bb, 30.

Brewer, E., 2012. CAP twelve years later: How the "rules" have changed. Computer 45 (2), 23–29.

de Wit, A., 2021. PCSE: The Python crop simulation environment. https://pcse.readthedocs.io.

de Wit, A., Boogaard, H., Fumagalli, D., Janssen, S., Knapen, R., van Kraalingen, D., Supit, I., van der Wijngaart, R., van Diepen, K., 2019. 25 years of the WOFOST cropping systems model. Agricult. Sys. 168, 154–167. http://dx.doi.org/10.1016/j.agsy.2018.06.018.

de Wit, A., Boogaard, H., Supit, I., van den Berg, M., 2020a. System Description of the WOFOST 7.2, Cropping Systems Model. Technical Report, Wageningen

Environmental Research, Publisher: Wageningen Environmental Research. URL https://edepot.wur.nl/522204.

de Wit, A., Boogaard, H., Supit, I., van den Berg, M. (Eds.), 2020b. System description of the WOFOST 7.2, cropping systems model, rev. 1.1 Wageningen Environmental Research.

de Wit, A., Hoek, S., Ballaghi, R., El Hairech, T., Qinghan, D., 2013. Building an operational system for crop monitoring and yield forecasting in Morocco. In: 2013 Second International Conference on Agro-Geoinformatics. Agro-Geoinformatics, pp. 466–469. http://dx.doi.org/10.1109/Argo-Geoinformatics.2013.6621964.

Dean, J., Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113. http://dx.doi.org/10.1145/1327452.1327492.

Flynn, M.J., 1966. Very high-speed computing systems. Proc. IEEE 54 (12), 1901–1909.

Flynn, M.J., 1972. Some computer organizations and their effectiveness. IEEE Trans. Comput. 100 (9), 948–960.

Fritz, S., See, L., Bayas, J.C.L., Waldner, F., Jacques, D., et al., 2019. A comparison of global agricultural monitoring systems and current gaps. Agricult. Sys. 168, 258–272.

Hack-ten Broeke, M., Mulder, H., Bartholomeus, R., van Dam, J., Holshof, G., Hoving, I., Walvoort, D., Heinen, M., Kroes, J., van Bakel, P., Supit, I., de Wit, A., Ruijtenberg, R., 2019. Quantitative land evaluation implemented in Dutch water management. Geoderma 338, 536–545. http://dx.doi.org/10.1016/j.geoderma.2018.11.002.

Hennessy, J.L., Patterson, D.A., 2011. Computer Architecture, Fifth Edition: A Quantitative Approach, fifth ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Huang, Y., xin CHEN, Z., YU, T., zhi HUANG, X., fa GU, X., 2018. Agricultural remote sensing big data: Management and applications. J. Integr. Agric. 17 (9), 1915–1931. http://dx.doi.org/10.1016/S2095-3119(17)61859-8.

Huang, J., Gómez-Dans, J.L., Huang, H., Ma, H., Wu, Q., Lewis, P.E., Liang, S., Chen, Z., Xue, J.-H., Wu, Y., Zhao, F., Wang, J., Xie, X., 2019. Assimilation of remote sensing into crop growth models: Current status and perspectives. Agricult. Forest. Meterol. 276–277, 107609.

Jang, W.S., Lee, Y., Neff, J.C., Im, Y., Ha, S., Doro, L., 2019. Development of an EPIC parallel computing framework to facilitate regional/global gridded crop modeling with multiple scenarios: A case study of the United States. Comput. Electron. Agric. 158, 189–200. http://dx.doi.org/10.1016/j.compag.2019.02.004.

Janssen, S., Porter, C., Moore, A., Athanasiadis, I., Foster, I., Jones, J., Antle, J., 2017. Towards a new generation of agricultural system data, models and knowledge products: Information and communication technology. Agricult. Sys. 155, 200–212.

Kim, J., Park, J., Hyun, S., Fleisher, D.H., Kim, K.S., 2020. Development of an automated gridded crop growth simulation support system for distributed computing with virtual machines. Comput. Electron. Agric. 169, 105196. http://dx.doi.org/10.1016/j.compag.2019.105196.

Kim, J., Park, J.Y., Hyun, S., Yoo, B.H., Fleisher, D.H., Kim, K.S., 2021. Development of an orchestration aid system for gridded crop growth simulations using kubernetes. Comput. Electron. Agric. 186, 106187. http://dx.doi.org/10.1016/j.compag.2021.106187.

van Kraalingen, D.W.G., Knapen, M.J.R., de Wit, A., Boogaard, H.L., 2020. WISS a java continuous simulation framework for agro-ecological modelling. In: Athanasiadis, I.N., Frysinger, S.P., Schimak, G., Knibbe, W.J. (Eds.), Environmental Software Systems. Data Science in Action. Springer International Publishing, Cham, pp. 242–248.

Kurtzer, G.M., Sochat, V., Bauer, M.W., 2017. Singularity: Scientific containers for mobility of compute. PLoS One 12 (5), e0177459.

Lahlou, M., CGMS-Maroc: National System for Agrometeorological Monitoring. 13. URL https://hdl.handle.net/20.500.11766/8881.

Lecerf, R., Ceglar, A., López-Lozano, R., Van Der Velde, M., Baruth, B., 2019. Assessing the information in crop model and meteorological indicators to forecast crop yield over europe. Agricult. Sys. 168, 191–202. http://dx.doi.org/10.1016/j.agsy.2018.03.002.

Li, Z., Qi, Z., Liu, Y., Zheng, Y., Yang, Y., 2023. A modularized parallel distributed high–performance computing framework for simulating seasonal frost dynamics in Canadian croplands. Comput. Electron. Agric. 212, 108057. http://dx.doi.org/10.1016/j.compag.2023.108057.

Liu, J., Koziol, Q., Butler, G.F., Fortner, N., Chaarawi, M., Tang, H., Byna, S., Lockwood, G.K., Cheema, R., Kallback-Rose, K.A., Hazen, D., Prabhat, M., 2018. Evaluation of HPC application I/O on object storage systems. In: 2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems. PDSW-DISCS, IEEE.

Parra-López, C., Abdallah, S., Garcia-Garcia, G., Hassoun, A., Sánchez-Zamora, P., Trollman, H., Jagtap, S., Carmona-Torres, C., 2024. Integrating digital technologies in agriculture for climate change adaptation and mitigation: State of the art and future perspectives. Comput. Electron. Agric. 226.

Paudel, D., de Wit, A., Boogaard, H., Marcos, D., Osinga, S., Athanasiadis, I.N., 2023. Interpretability of deep learning models for crop yield forecasting. Comput. Electron. Agric. 206, 107663. http://dx.doi.org/10.1016/j.compag.2023.107663.

Poggio, L., de Sousa, L.M., Batjes, N.H., Heuvelink, G.B.M., Kempen, B., Ribeiro, E., Rossiter, D., 2021. SoilGrids 2.0: producing soil information for the globe with quantified spatial uncertainty. SOIL 7 (1), 217–240. http://dx.doi.org/10.5194/soil-7-217-2021.

Porter, C., Villalobos, C., Holzworth, D., Nelson, R., White, J., Athanasiadis, I., Janssen, S., Ripoche, D., Cufi, J., Raes, D., Zhang, M., Knapen, M., Sahajpal, R., Boote, K., Jones, J., 2014. Harmonization and translation of crop modeling data to ensure interoperability. Environ. Model. Softw. 62, 495–508.

Ross, K.W., Brown, M.E., Verdin, J.P., Underwood, L.W., 2009. Review of FEWS NET biophysical monitoring requirements. Environ. Res. Lett. 4 (2), 024009. http://dx.doi.org/10.1088/1748-9326/4/2/024009.

Rosser, B., 1941. Alonzo Church. The calculi of lambda-conversion. Annals of Mathematics studies, no. 6. Lithoprinted. Princeton University Press, Princeton1941, 77 pp. J. Symb. Log. 6 (4), 171–172. http://dx.doi.org/10.2307/2267126.

Roy, P.V., Haridi, S., 2004. Concepts, Techniques, and Models of Computer Programming. MIT Press.

Siddiqa, A., Karim, A., Gani, A., 2017. Big data storage technologies: a survey. Front. Inf. Technol. Electron. Eng. 18 (8), 1040–1070.

U.S. Department of Agriculture, 2012. The yield forecasting program of NASS. URL https://www.nass.usda.gov/Education_and_Outreach/Understanding_Statistics/Yield_Forecasting_Program.pdf.

Vakhtang, S., James, H., Vaishali, S., Cheryl, P., Pramod, A., Carol J., W., Gerrit, H., 2019. A multi-scale and multi-model gridded framework for forecasting crop production, risk analysis, and climate change impact studies. Environ. Model. Softw. 115, 144–154. http://dx.doi.org/10.1016/j.envsoft.2019.02.006.

van der Velde, M., van Diepen, C., Baruth, B., 2019. The European crop monitoring and yield forecasting system: Celebrating 25 years of JRC MARS bulletins. Agricult. Sys. 168, 56–57. http://dx.doi.org/10.1016/j.agsy.2018.10.003.

Whitcraft, A., Becker-Reshef, I., Justice, C., 2020. NASA harvest(ing) earth observations for informed agricultural decisions. In: IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium. pp. 3706–3708. http://dx.doi.org/10.1109/IGARSS39084.2020.9324176.

Wolfert, S., Ge, L., Verdouw, C., Bogaardt, M., 2017. Big data in smart farming – a review. Agricult. Sys. 153, 69–80.