Mitigating synchronization bottlenecks in high-performance actor-model-based software

Kyle Klenk
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
kyle.klenk@usask.ca

Mohammad Mahdi Moayeri
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
m.moayeri@usask.ca

Junwei Guo
Department of Civil Engineering
University of Calgary
Calgary, Canada
junwei.guo@ucalgary.ca

Martyn P. Clark
Department of Civil Engineering
University of Calgary
Calgary, Canada
martyn.clark@ucalgary.ca

Raymond J. Spiteri
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
spiteri@cs.usask.ca

***Abstract*—Bulk synchronous programming (in distributed-memory systems) and the fork-join pattern (in shared-memory systems) are often used for problems where independent processes must periodically synchronize. Frequent synchronization can greatly undermine the performance of software designed to solve such problems. We use the actor model of concurrent computing to balance the load of hundreds of thousands of short-lived tasks and mitigate synchronization bottlenecks by buffering communication via actor batching. The actor model is becoming increasingly popular in scientific and high-performance computing because it can handle heterogeneous tasks and computing environments with enhanced programming flexibility and ease relative to conventional paradigms like MPI. For a hydrologic simulation of continental North America with over 500,000 elements, the proposed buffering approach is approximately 4 times faster than no buffering, outperforms MPI on single and multiple nodes, and remains competitive with OpenMP on a single node and MPI+OpenMP on multiple nodes.**

***Index Terms*—actor model of concurrent computing, high-throughput computing, scientific and high-performance computing, bulk synchronous programming, fork-join pattern**

I. Introduction

The actor model of concurrent computing is increasingly being applied with success to problems in high-performance computing (HPC) [1]–[4]. This success stems from the simplicity of expressing concurrency with the actor model, making it an excellent fit for problems that use bulk synchronous parallel (BSP), fork-join, or similar patterns. Typically, actor-model implementations, such as the C++ Actor Framework (CAF) [5], employ a work-stealing algorithm that is well-suited for these patterns.

However, the performance of the actor model can be hindered in computations that require frequent synchronization of hundreds of thousands of short-lived tasks. This is especially true when using the conventional approach of defining an actor for each task, with a single actor responsible for synchronization. In this paper, we show how to mitigate this performance bottleneck by introducing an additional layer of actors to buffer communication and reduce contention on the synchronization actor. We compare the performance of this approach with the conventional actor model, MPI, and OpenMP implementations using single-node experiments, and to MPI and MPI+OpenMP implementations using multi-node experiments.

The BSP or fork-join patterns on which we focus in this study is that induced by the requirement of *data assimilation*. Data assimilation is the common practice in simulation science of incorporating observational data into computational models in order to improve their predictive power. It has been used for decades in meteorology and hydrology, and interest in it has further increased recently with the incorporation of machine learning and artificial intelligence in simulations, e.g., for creating digital twins, e.g., [6]–[8].

A simulation involving data assimilation generally consists of the evolution of many state variables, which may vary from tightly coupled to completely uncoupled (embarrassingly parallel). At the synchronization points, observations are input, and the state variables are adjusted to account for the new information provided by the data. Accordingly, simulations of all variables must wait at the synchronization points until they have all reached it before they can proceed. This pattern is called BSP when it occurs in a distributed-memory system or fork-join when it occurs in a shared-memory system.

This study aims to mimic the process of data assimilation in a hydrologic simulation over continental North America. The simulation is performed using the SUMMA-Actors [9], SUMMA-MPI, SUMMA-OpenMP, and SUMMA-MPI+OpenMP models, which are all based on the SUMMA (Structure for Unification of Multiple Modeling Alternatives) land model [10]. The focus of this study is to compare the performance of the actor model against the traditional parallel programming paradigms of MPI, OpenMP, and MPI+OpenMP in the context of data assimilation, while proposing a solution to mitigate synchronization bottlenecks that can occur when using the actor model.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00168

SUMMA characterizes the spatial domain in terms of two hierarchical elements: grouped response units (GRUs) and hydrological response units (HRUs). GRUs and HRUs combine to represent a landscape that comprises the simulation domain. GRUs are the higher-level element and contain one or more HRUs. GRUs can be of any shape and size, but they must be spatially contiguous. Similarly, HRUs can be of any shape and size, but each must fit entirely within a GRU. HRUs are defined to be spatial areas that have relatively homogeneous hydrologic properties and hence may not be spatially contiguous. Their main use is to capture sub-scale variability within a GRU. SUMMA treats the GRUs of a given simulation independently. In this study, all GRUs contain exactly one HRU.

The usual way of running SUMMA is to run all GRUs associated with a spatial domain in an embarrassingly parallel fashion over a predefined number of *data windows* (timesteps) until they all reach a final end time. We refer to this way of performing a simulation as *asynchronous mode*. In order to mimic data assimilation, we impose the constraint that all GRUs must complete their data window before any computations on the next data window can begin. We refer to this way of performing a simulation as *data-assimilation mode*.

Data-assimilation mode within SUMMA can be hard to manage using traditional programming paradigms such as MPI. Besides the high communication costs, which arguably are part of the problem statement, the main challenge is that the execution times of the GRUs on a given data window are variable, and furthermore this variability also changes depending on the data window. Accordingly, there is no realistic way to load-balance the computations a priori to minimize the straggler effect. In contrast, OpenMP can be viewed as a more natural fit for data assimilation because it can easily handle the fork-join pattern, but then it requires the use of MPI to synchronize across nodes. The actor model on the other hand, can handle the parallelism and synchronization both on a single node and across nodes in a single framework.

The actor model has been increasingly emerging as a performant and straightforward computational model for high-performance computing. It offers a high-level computational abstraction known as an *actor*, which developers can use to express concurrency in a way that is easy to understand and reason about. actor-model implementations, such as the C++ Actor Framework (CAF) [5] and Akka [11], typically offer a straightforward syntax for defining actors and their interactions, and this facility can reduce the cognitive burden on developers who find themselves creating increasingly elaborate applications.

Generally, under the hood of actor-model implementations is a work-stealing algorithm that effectively manages the concurrent execution of actors. Work-stealing algorithms typically keep tasks/actors from migrating between the threads/cores in order to minimize cache misses between tasks/actors that interact frequently [12]. In [9], the effectiveness of SUMMA-Actors was demonstrated to improve the resource utilization and fault tolerance of SUMMA run in asynchronous mode. However, the fork-join model can severely impact performance when a single actor serves as a synchronization point for hundreds of thousands of actors that execute short-duration tasks, such as the simulation of many GRUs over short data windows. In order to mitigate this issue, we judiciously assemble a hierarchy of actors that act to buffer communication between many actors. Accordingly, we can create significant performance gains by reducing contention on a single actor that is responsible for the synchronization of the entire program.

We demonstrate how to apply the actor model with a scheduler based on a general work-stealing algorithm to mitigate synchronization bottlenecks and hence improve code performance while providing a comprehensive comparison of the performance against MPI, OpenMP, and MPI+OpenMP. The contribution of this study is three-fold. First, we demonstrate a new application of the actor model to a common issue in HPC, namely that of synchronization bottlenecks. We also show how software designed to solve such problems actually falls loosely into the category of the actor model, and more specifically, how such software may actually benefit from considering the point of view of the actor model. Second, we explicitly demonstrate performance improvements from applying the actor model to problems that rely on the BSP or fork-join patterns, specifically in the context of data assimilation applied to a large hydrological simulation. Third, we provide a detailed comparison of the performance of the actor model against MPI, OpenMP, and MPI+OpenMP implementations of SUMMA.

The rest of this paper is organized as follows. In section II, we describe related work in the field of improving load balancing and mitigating synchronization bottlenecks as well as give some background on the actor model, the SUMMA implementations, and a thorough introduction to the actors-based code to handle data-assimilation mode. In section III, we describe the methodology of our study and the experiments performed. In section IV, we describe the results from the experiments and provide some discussion. Finally, in section V, we summarize our findings and conclusions and offer a few directions for future work.

II. Background

A. Related Work

A defining aspect of the BSP and fork-join patterns is that they introduce a synchronization point in the execution of a program. The presence of synchronization points can significantly reduce performance. Profiling optimization techniques [13] and lock-free synchronization mechanisms [14] can help reduce synchronization bottlenecks and contention at synchronization points. However, utilizing these techniques often requires a deep understanding and even re-evaluation of the software, potentially increasing complexity and complicating the debugging process [14]. Event-driven programming models can also handle synchronization bottlenecks implicitly via event loops that process events in a non-blocking manner [15]. The actor model falls into this category of an event-

driven programming model, where actors are coordinated and synchronized by the messages (events) they send to each other. Nonetheless, synchronization bottlenecks can still occur within the actor model, especially when many actors synchronize with a single actor, as is typical in the BSP and fork-join patterns.

One specific area of interest that has emerged in the HPC community is the development of a programming model known as partitioned global address spaces (PGAS) and its application to problems centered around the BSP and fork-join patterns. PGAS models provide a global and coherent view of memory across different nodes in a distributed system [16]. This model is characterized by a global memory address space that is logically partitioned, where each partition is local to a processing element. The key feature of PGAS is a process can directly access memory with affinity to a different, potentially remote process without the explicit involvement of the target process. However, the PGAS model is not immune to synchronization bottlenecks. The work carried out in [17] addressed this issue by borrowing some principles from the actor model, including mailbox data structures and termination detection to improve the performance of BSP applications. A key element of the approach was the utilization of *conveyors* [18] to automatically aggregate messages. Mixing concepts within the context of PGAS and the actor model is not new and has been explored in implementations such as ActorX10 [1] and Actor-UPC++ [19]. The novelty of our batching approach, however, is that there is no mixing of programming models; rather, the actor model itself can be used to mitigate synchronization bottlenecks in a BSP or fork-join pattern. This reduces the complexity arising from combining various concepts that can burden programmer productivity and challenge software maintainability.

Similarly, Shiina and Taura introduced a new task-parallel runtime system called Itoyori to increase the performance of global fork-join parallelism in distributed computing environments [12] by integrating efficient cache sharing within the combination of PGAS and a task-parallel runtime system. In the current study, we also focus on performance improvements, but we target synchronization bottlenecks specifically in the context of the actor model. We demonstrate how our approach can be applied across multiple nodes in a distributed environment. We also compare the performance of the actor model against MPI, OpenMP, and MPI+OpenMP, none of which seem to be addressed in the literature.

Another fundamental component of addressing synchronization bottlenecks and inefficiencies in BSP and fork-join patterns within the context of the actor model and task-based runtime systems is the underlying scheduler. Systems such as Legion [20] and PaRSEC [21] offer parallelism by focusing on tasks as the primary unit of computation and communication. Legion is a data-centric programming system that automatically optimizes data movement and task execution for the underlying hardware architecture. PaRSEC is a task-based runtime system that emphasizes dynamic scheduling of fine-grained tasks on distributed, heterogeneous architectures by utilizing direct acyclic graphs to represent computational tasks and their data dependencies.

On the other hand, dynamic load-balancing strategies, such as work-stealing and off-loading in parallel computing systems [22], attempt to efficiently distribute computational tasks among multiple processors or threads [23]. Work-stealing aims to minimize idle time and ensure that all processing units are equally engaged with useful work by allowing idle processors to take tasks from those with excess tasks in their queues. Different work-stealing strategies have been developed for actor-based frameworks. For example, Actor-UPC++ offers fully asynchronous, diffusion-based load-balancing strategies, including global and local versions of actor stealing and off-loading, to improve load distribution, performance, and efficiency [22]. Charm++ [24] is a C++ parallel programming framework with a computational model similar to the actor model. Charm++ decomposes the program into a number of cooperating message-driven objects known as *chares*. The Charm++ runtime system offers a variety of dynamic load-balancing strategies, such as persistence-based, communication- and topology-aware, and work-stealing. PackStealLB is a notable example of a work-stealing load balancer in Charm++ [25]. It is a distributed load balancer that combines the *packing* technique with constrained and randomized work-stealing heuristics. Another innovative approach within Charm++ is distributed work-stealing via matchmaking [26]. The matchmaking scheduler efficiently pairs idle computing nodes with those overloaded with tasks, operating with low overhead. This scheduler employs a centralized or distributed matchmaker that keeps track of both the availability of tasks and the demand for tasks across the system. However, in this study, we specifically highlight the importance of the underlying scheduler, but in the spirit of reducing the cognitive burden on the programmer, we offer a solution that does not require specific modification of the scheduler itself to increase performance. Instead, we show how an additional layer of actors to buffer communication and reduce contention on single actors can be introduced within an actor-model implementation that uses a work-stealing scheduler.

B. Actor Model

The actor model is a computational model where users compose their programs as a set of actors whose interactions are defined in terms of messages [27]. This model is increasingly being applied to problems in HPC because it is conceptually easy to understand and has great potential to fit the structure of many programs well while providing a high degree of performance. The actor model is also highly regarded as a safe model for concurrent programming that is inherently free from race conditions and deadlocks [28]. This inherent safety stems from the fact that actors are independent entities with no direct access to each other's state. Instead, actors communicate via asynchronous message passing. Typically, actor-model implementations incorporate a mailbox data structure where messages are stored until they are processed sequentially by the receiving actor [29]. Because actors are independent and use message-passing semantics, they can

easily be extended into distributed systems, allowing actors to easily communicate across different nodes in a network.

One of the most important parts of an actor-model implementation is the scheduler, which is responsible for managing the execution of actors. The scheduler is usually transparent to the user, allowing the user to focus on the logic and structure of actors in their program. Different actor-model implementations offer different scheduling strategies, and some even offer multiple strategies from which the user can select. Furthermore, some actor-model implementations encourage users to implement their own custom scheduling strategies. A popular choice for the scheduling strategy in actor-model implementations is work-stealing. Work-stealing was a scheduling technique first proved to be efficient in multithreaded programs by Blumofe and Leiserson [30]. Today, it serves as the foundation for many actor-model implementations, including CAF [5], Akka [11], Pony [31], and React++ [32], to name a few. Moreover, work-stealing is still a popular choice in general for scheduling tasks for multi-core systems as well as distributed systems through a hierarchical work-stealing design [33].

In this study, we observe that CAF's work-stealing scheduler can become a bottleneck when executing a BSP or fork-join pattern with hundreds of thousands of actors attempting to communicate with a single actor frequently. Although we observe this behavior in CAF, we expect that it is a general issue that can occur in any actor-model implementation that uses a work-stealing scheduler.

C. SUMMA

SUMMA [34] is a hydrological model (a so-called *land model*) that is used to simulate the thermodynamics and hydrology of a given geographic region. SUMMA was designed to be a framework that can be conveniently used to represent and compare different theories and representations of hydrological processes. Figure 1 depicts the domains and primary fluxes represented in SUMMA.

Currently, there are four different implementations of parallelization for SUMMA: actors, MPI, OpenMP, and MPI+OpenMP. Each of these implementations is explained in the subsequent sections.

1) SUMMA-Actors: Recently, SUMMA was parallelized using the actor model via CAF, resulting in a code called SUMMA-Actors. SUMMA-Actors successfully increased the efficiency and fault tolerance of SUMMA and delivered superior performance on HPC systems [9]. However, the implementation of SUMMA-Actors is limited to the independent and asynchronous simulation of GRUs through its implementation of the `hru_actor` (now `gru_actor`). This method is not suitable for real-time data analysis, where the objective is to simulate all input data for a given data window before the arrival of data for the next data window.

The initial design of SUMMA-Actors partitioned SUMMA into a hierarchy of actors, each responsible for a different part of the computation (see figure 2). This design provides a supervision structure for increased fault tolerance such that a parent actor is responsible for supervising the computation of its child

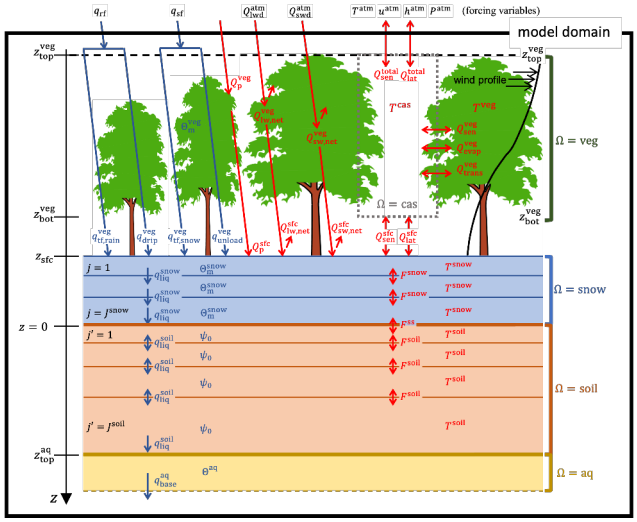

Fig. 1. Schematic of the SUMMA land model.

actors. At the top of the hierarchy is the `summa_actor`, which is primarily a supervisor actor, but it also provides the ability for the program to be sub-batched as desired, e.g., when the simulation is too large to fit into memory of a single system. Although not depicted in the figure, the `summa_actor` can be connected to a `summa_client_actor` and a `summa_server_actor` for use in or for creation of ad-hoc distributed computing environments [35]. The `job_actor` is the child of the `summa_actor` and is responsible for creating and managing the `gru_actors` that carry out the simulation of the GRUs. The `job_actor` also creates and supervises the `file_access_actor`, which handles all file I/O operations.

2) SUMMA-MPI: The SUMMA-MPI implementation separates the simulation of the GRUs into separate MPI processes. When a simulation starts, SUMMA-MPI evenly distributes the GRUs among the MPI processes. Each MPI process is then responsible for simulating its batch of GRUs, including parallel file reading and writing. To fufill the requirements of data-assimilation mode, SUMMA-MPI uses a barrier to synchronize the MPI processes at the end of each data window.

3) SUMMA-OpenMP: The SUMMA-OpenMP implementation is the most straightforward of the four implementations. It uses OpenMP to parallelize the simulations of GRUs by placing directives before the GRU simulation loop and within the data window loop. With the above implementation, SUMMA-OpenMP meets the requirements of data-assimilation mode using a fork-join pattern. For each data window, the input data are read, and the simulation of each GRU is performed in parallel. Once all GRUs have completed their data window, their results are written, and the next data window is started.

4) SUMMA-MPI+OpenMP: The last implementation of SUMMA is a hybrid approach using both MPI and OpenMP to construct SUMMA-MPI+OpenMP. This implementation uses

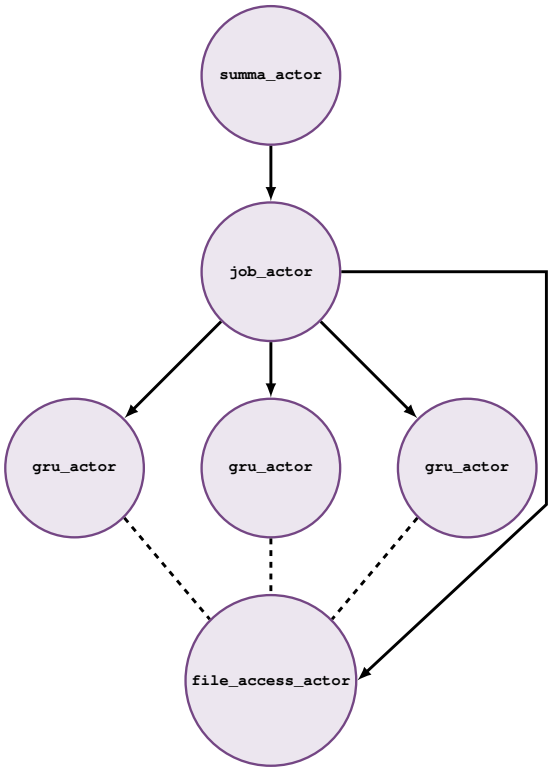

Fig. 2. The initial Summa-Actors program and the diagram of Summa-Actors in data assimilation mode with no batch_actors. Since its first publication, SUMMA-Actors has replaced the hru_actor with the gru_actor to handle scenarios where a GRU can contain multiple HRUs. This figure has been updated to reflect this change.

MPI in the same way as section II-C2 to distribute the GRUs among MPI processes, and uses a barrier to synchronize the MPI processes at the end of each data window. Each MPI process then uses OpenMP to parallelize the simulation of its batch of GRUs within each data window. This hybrid implementation should provide better performance than SUMMA-MPI because it can balance the load of the GRUs dynamically and offers the ability to run in distributed environments which SUMMA-OpenMP cannot do on its own.

D. SUMMA-Actors: Data-Assimilation Mode

To implement data-assimilation mode in SUMMA-Actors, it is required that all gru_actors complete a given data window before moving onto the next one. An obvious method to implement this requirement is to modify the job_actor to coordinate all gru_actors. The job_actor would send a start message to each gru_actor and then wait for all gru_actors to send a completion message before starting them on the next data window. Although fine in theory, this strategy quickly leads to a significant bottleneck in practice for large problems. Because the job_actor is responsible for coordination, it becomes overloaded when a large number of gru_actors complete at essentially the same time. This situation occurs when the simulation of each GRU's data window is short-lived (and hence effectively uniform), and typically the run-time is dominated by the sheer number of GRUs rather than their computational difficulty. For continental domains, the number of GRUs is in the hundreds of thousands. However, in asynchronous mode (or when many data windows are computed between synchronizations), which is the use case for which SUMMA-Actors was designed, no communication bottlenecks are observed because the job_actor only interacts with the gru_actors at the start and end of their simulations, and these events are well spaced in time.

To alleviate this bottleneck, we implement an additional layer of actors to the hierarchical design of SUMMA-Actors. Instead of the job_actor spawning and managing individual gru_actors, it spawns and manages several batch_actors, which in turn spawn and manage individual gru_actors. This hierarchy of actors is shown in figure 3. The intent of the batch_actor is to reduce the contention on the job_actor and increase the efficiency of CAF's work-stealing scheduler. The efficiency gains are provided by the fact that the batch_actor is more likely to have it and its children's tasks scheduled on the same thread/core. To ensure this is the case, the number of batch_actors is set to the number of cores on the system at runtime. Furthermore, the number of messages the job_actor receives is significantly reduced because now it only interacts with as many batch_actors as there are CPU cores.

This application of an additional layer of actors for the purpose of batching provides several advantages over other potential solutions. First, the approach is general and can be applied to a wide range of actor-model programs or task-based parallel programs that exhibit a fork-join or similar pattern where the parallel tasks are short-lived. The generality lies in the ability for the user to implement their own specific form of batch actor and apply it to their program. Second, the method has a hierarchical design that enables users to customize the batching process according to their program's demands. For instance, our application used a single layer of batch_actors to manage over 500,000 GRUs. Should the number of GRUs increase such that the initial batch actor becomes insufficient, however, users have the flexibility to add further batching layers. This flexibility helps to efficiently manage significant changes in workload. Third, the approach is straightforward and requires minimal and non-invasive changes to the existing code. Fourth, this solution does not require complicated analysis, tuning of scheduler implementations, or the use of software packages in addition to what is already being used.

This initial approach to data-assimilation mode, however, is unable to span multiple nodes. To address this limitation, we added another mode of operation by modifying the hierarchy of actors within SUMMA-Actors. The structure of this mode is provided in figure 4. To achieve data-assimilation mode in distributed environments, we replaced the top-level summa_actor with the da_server_actor and the job_actor with a da_client_actors that reside on each node. These two new types of actors better align with

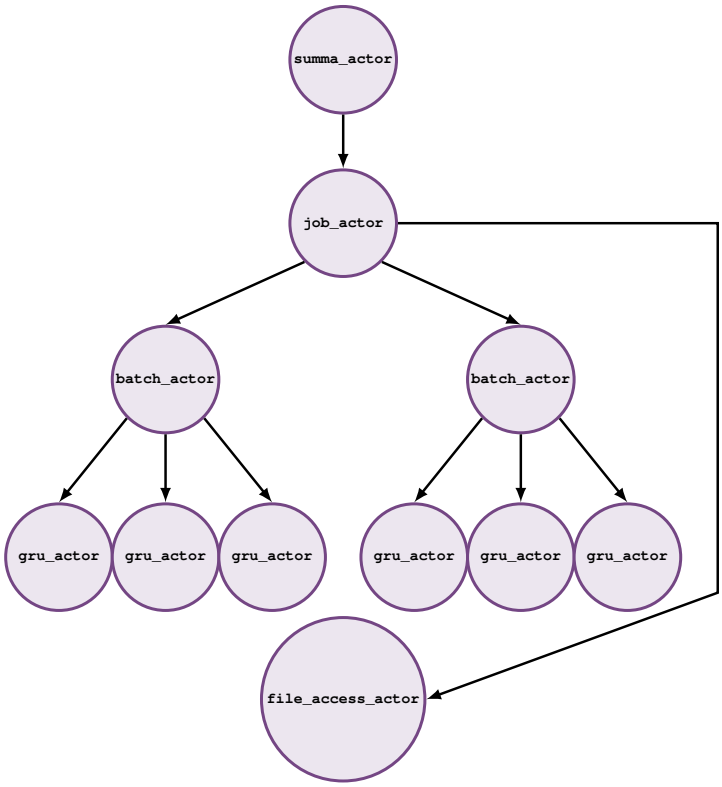

Fig. 3. The SUMMA-Actors application in data-assimilation mode with the addition of `batch_actor`s.

the task of distributing the simulation across multiple nodes.

In distributed data-assimilation mode, SUMMA-Actors starts by spawning the `da_server_actor` on the first node. The `da_server_actor` then spawns its own local `da_client_actor`, reducing the need for a separate node just for the `da_server_actor`. The user specifies how many nodes are to take part in the simulation, and the `da_server_actor` waits for all `da_client_actor`s to connect before starting the simulation. Once all `da_client_actor`s have connected, the `da_server_actor` sends a start message to each `da_client_actor` and then waits for all `da_client_actor`s to send a completion message before moving onto the next data window. Each `da_client_actor` then spawns and manages several `batch_actor`s, which in turn spawn and manage individual `gru_actor`s.

III. Methodology

To investigate the effect a single layer of `batch_actor`s has on SUMMA-Actors throughput in data-assimilation mode, we performed several hydrological simulations over continental North America. We then compared the results to SUMMA-Actors without a `batch_actor`, SUMMA-MPI, and SUMMA-OpenMP in three single-node experiments. For a multi-node experiment, we compared SUMMA-Actors with a `batch_actor` to SUMMA-MPI and SUMMA-MPI+OpenMP. The domain used consists of 517,315 GRUs with forcing data from January, 2019. The forcing data were obtained from the workflow described in [36] and are publicly available at https://zenodo.org/records/13687422.

The three single-node experiments were performed on the `Copernicus` cluster located at the University of Saskatchewan in Saskatchewan, Canada. `Copernicus` consists of 50 nodes, each with various CPU and GPU configurations. The nodes used for the simulations consisted of dual-socket Intel Xeon Gold 6142 (Skylake) processors, totaling 32 cores and 326 GB of RAM each. The nodes were connected to a shared file system using a 100 Gbps Infiniband network.

The fourth experiment was a multi-node experiment performed on the `Niagara` cluster located at the University of Toronto in Ontario, Canada. `Niagara` is part of the Digital Research Alliance of Canada[1] and consists of 2024 nodes, each with either 40-core Intel Skylake or 40-core Intel Cascade Lake processors and 202 GB of RAM. The nodes were connected to a shared file system using an EDR Infiniband network.

The experiments performed were

E1: a one-day simulation (24 hourly data windows) executed on a single node,

[1]https://alliancecan.ca/en

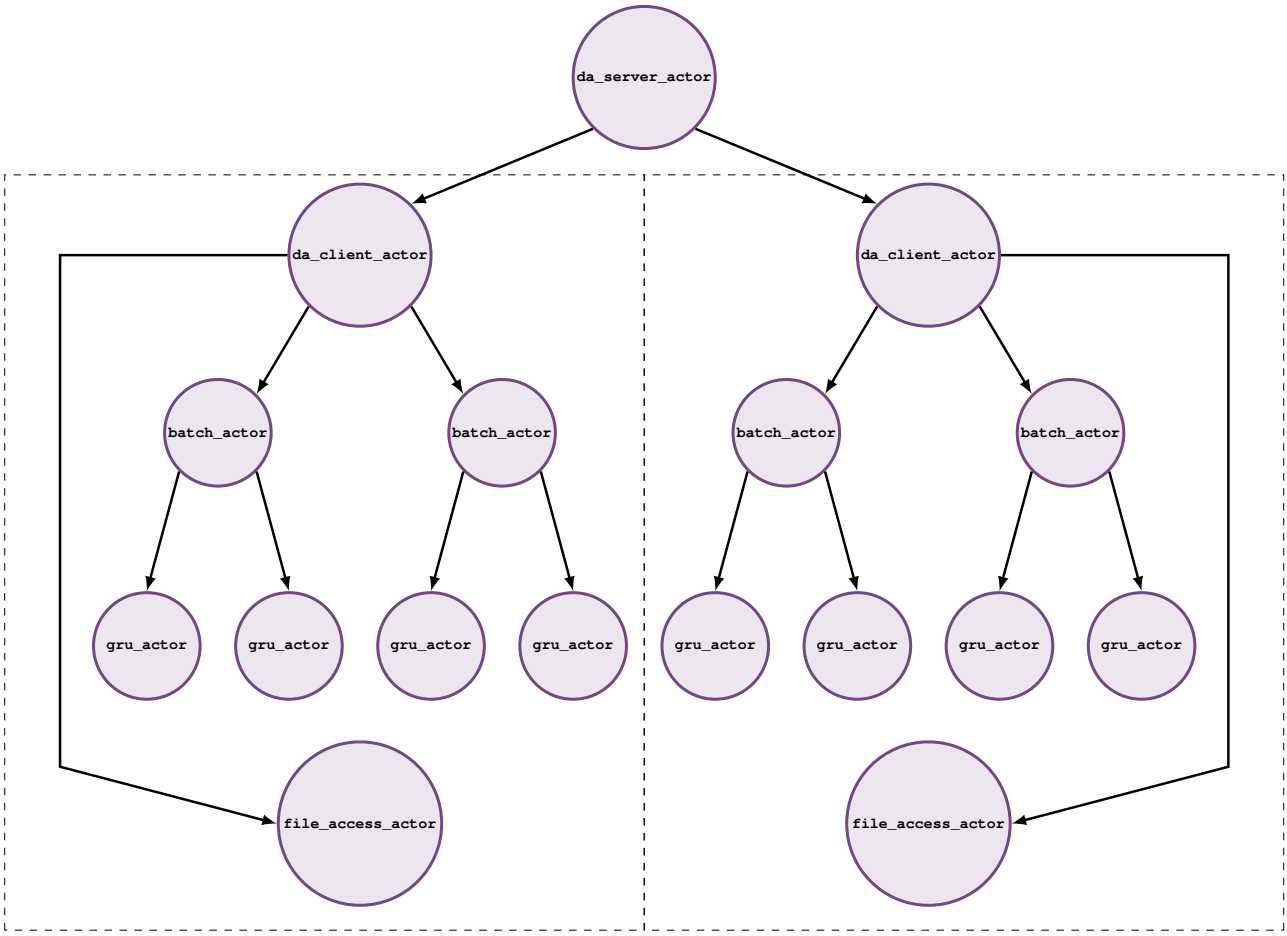

Fig. 4. The actors involved in running data-assimilation mode in a distributed environment. The dashed lines delineate the nodes in the distributed environment. The `da_server_actor` resides on one of the nodes.

E2: a one-week simulation (168 hourly data windows) executed on a single node,
E3: a one-month simulation (744 hourly data windows) executed on a single node, and
E4: a one-month simulation (744 hourly data windows) executed on 10, 20, and 40 nodes.

The first three experiments consisted of running the following implementations and configurations of SUMMA:

1) SUMMA-Actors in data-assimilation mode with batching,
2) SUMMA-Actors in data-assimilation mode with no batching,
3) SUMMA-MPI with a barrier to synchronize the MPI processes at the end of each data window, and
4) SUMMA-OpenMP.

For Experiment 4, we investigated the scaling behavior of the following implementations and configurations of SUMMA:

1) SUMMA-Actors in data-assimilation mode with batching,
2) SUMMA-MPI with a barrier to synchronize the MPI processes at the end of each data window, and
3) SUMMA-MPI+OpenMP.

Each SUMMA implementation was run on 10, 20, and 40 nodes. Although each node on `Niagara` has 40 cores, hyperthreading was enabled, so each job utilized 80 logical cores per node.

IV. Results and Discussion

A. Experiment 1: one-day simulation, single node

In Experiment 1, we simulated the hydrology over continental North America for one day (24 hourly data windows) using a single node. The results of this experiment are shown in figure 5. In the top chart of the figure, we observe the duration of each job as measured by Slurm. In the bottom chart, the total GRU physics durations measured by SUMMA are displayed as bars. Additionally, 1000 samples of the GRU physics durations are plotted as dots within their respective bars to verify the accuracy of the original measurements. Each sample was taken by randomly selecting 517,315 GRUs with replacement

and summing their physics durations. The durations for each GRU were obtained using Fortran's `system_clock` intrinsic function. Each measurement was taken at the beginning and end of SUMMA's `coupled_em` subroutine, which is the same for all implementations. The values in the bottom chart are then achieved by summing all measured GRU timings.

Beginning with the top chart, we observe that SUMMA-OpenMP required the least time to complete its job at 0.69 hours. This was followed by SUMMA-Actors with batching at 0.76 hours, SUMMA-MPI at 0.78 hours, and SUMMA-Actors without batching at 3.10 hours. Moving to the bottom chart, we find the ranking changes, with SUMMA-MPI spending the least amount of time computing the GRU physics at 18.37 hours, followed by SUMMA-OpenMP at 18.77 hours, SUMMA-Actors without batching at 20.37 hours, and SUMMA-Actors with batching taking the most time at 20.44 hours. Within each bar in the bottom chart, we see each of the 1000 samples of the GRU physics durations are tightly clustered around their original measurement. This indicates little variability in the GRU physics durations, suggesting that the original measurements are reliable.

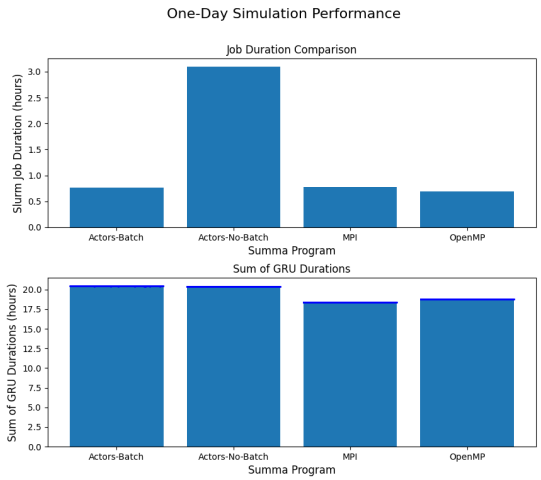

Fig. 5. Comparison of the job durations for one-day simulations (24 hourly data windows) as measured by Slurm (top chart) and the sum of the GRU physics durations as measured by SUMMA (bottom chart). Dots within the bottom bars represent the samples of the GRU physics durations taken using the bootstrapping method.

Analyzing the results of Experiment 1, we see that the `batch_actor` in SUMMA-Actors makes a significant difference in performance, marking a 121.2% improvement in job duration. This improvement is likely due to reduced contention on the `job_actor` because multiple `batch_actors` handle communication with groups of `gru_actors`, buffering the communication before an update to the `job_actor`. The `batch_actors` are more likely to have their tasks scheduled on the same CPU core, reducing the chance of migration between CPUs. Without the `batch_actor`, the `job_actor` will migrate constantly between CPU cores as they become available for it to service the updates from the `gru_actors`, leading to increased contention and poor cache utilization.

This batching approach did not outperform SUMMA-OpenMP, which completed 9.6% faster, but the batching approach does outperform SUMMA-MPI by 2.5%. SUMMA-MPI encounters the straggler effect, where processes that finish their GRUs early must wait for others to complete before proceeding to the next data window. Accordingly, the applications that can balance the GRUs across CPUs are expected to perform better than SUMMA-MPI.

To better understand how SUMMA-OpenMP outperforms SUMMA-Actors with batching, we turn to the bottom chart. We find that although SUMMA-OpenMP is the second most efficient at computing the GRU physics, it balances the GRUs across CPUs more effectively, enabling it to complete the simulation in the least amount of time. SUMMA-Actors with batching also balances the GRUs across CPUs well, allowing it to outperform SUMMA-MPI. However, it suffers from slower physics computations compared to SUMMA-OpenMP, which enables SUMMA-OpenMP to complete the simulation faster.

Both SUMMA-Actors simulations show a reduction of performance in computing the GRU physics compared to SUMMA-MPI and SUMMA-OpenMP. This reduction is likely due to the overhead of using Fortran and C++ to enable the actor model in SUMMA-Actors. SUMMA-MPI and SUMMA-OpenMP are written exclusively in Fortran, which likely allows for better compiler optimization and cache utilization. However, SUMMA-Actors must use C++ wrappers to call Fortran code, leading to increased overhead and less efficient data structures.

The difference in GRU physics performance between SUMMA-OpenMP and SUMMA-Actors with batching is 8.5%, suggesting that without the increased overhead of using two programming languages, SUMMA-Actors with batching would perform closer to SUMMA-OpenMP in overall job duration.

B. Experiment 2: one-week simulation, single node

In Experiment 2, the simulation was extended in duration to one week (168 hourly data windows) in anticipation that the implementations of SUMMA that have some form of load balancing would perform better. The results of this experiment are shown using the same format as Experiment 1 in figure 6.

Observing the results shown in the top chart, we again see the same order of performance as in Experiment 1. SUMMA-OpenMP required the least amount of time to complete its job at 4.17 hours, followed by SUMMA-Actors with batching at 4.46 hours, SUMMA-MPI at 4.72 hours, and SUMMA-Actors with no batching at 20.70 hours. These results tell largely the same story in terms of job duration as Experiment 1. The gap between SUMMA-OpenMP and SUMMA-Actors with batching was closed slightly, with their difference being 6.7%, compared to 9.6% in Experiment 1. Both SUMMA-OpenMP and SUMMA-Actors had a larger gap between them and SUMMA-MPI than in Experiment 1, with the difference being

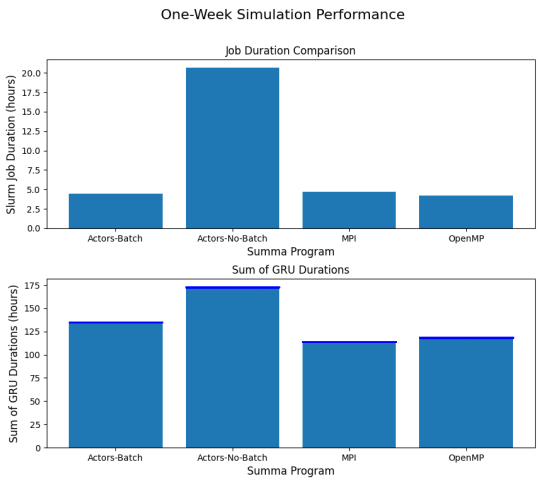

Fig. 6. Comparison of the job durations for a one-week simulation (168 hourly data windows) as measured by Slurm (top chart) and the sum of the GRU physics durations as measured by SUMMA (bottom chart). Dots within the bottom bars represent the samples of the GRU physics durations taken using the bootstrapping method.

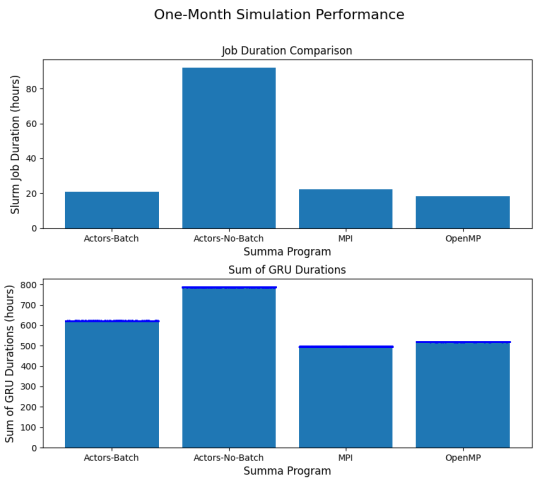

Fig. 7. Comparison of the job durations for a one-month simulation (744 hourly data windows) as measured by Slurm (top chart) and the sum of the GRU physics durations as measured by SUMMA (bottom chart). Dots within the bottom bars represent the samples of the GRU physics durations taken using the bootstrapping method.

12.3% and 5.6%, respectively. This again is because of the straggler effect encountered by SUMMA-MPI.

In the bottom chart of figure 6, we again see the same order of performance as in Experiment 1. SUMMA-MPI required the least amount of time to compute the GRU physics at 114.33 hours, followed by SUMMA-OpenMP at 119.62 hours, SUMMA-Actors in data-assimilation mode with batching at 137.51 hours, and SUMMA-Actors in data-assimilation mode with no batching at 172.75 hours. Similarly, the samples of the GRU physics durations are tightly clustered around their original measurements, indicating that the original measurements are reliable.

The same factors as outlined in Experiment 1 likely contributed to the results of this experiment. However, SUMMA-Actors without batching performed considerably worse than SUMMA-Actors with batching in terms of GRU physics duration in this experiment than in the previous one. The longer simulation duration likely exacerbated the contention on the `job_actor`, leading to increased overhead and even worse cache utilization.

C. Experiment 3: one-month simulation, single node

In Experiment 3, we further extended the duration of the simulation to one month (744 hourly data windows). This experiment should again benefit the SUMMA implementations with load balancing. As in the previous two experiments, the results of this experiment are shown in the same format in figure 7. For this experiment, the order of job durations remains the same, with SUMMA-OpenMP requiring the least amount of time at 18.21 hours, followed by SUMMA-Actors with batching at 20.81 hours, SUMMA-MPI at 22.25 hours, and SUMMA-Actors with no batching taking 92.05 hours.

Looking at the bottom chart, the story is the same as the previous two experiments, with SUMMA-MPI spending the least amount of time computing the GRU physics at 495.94 hours, followed by SUMMA-OpenMP at 518.08 hours, SUMMA-Actors with batching at 622.23 hours, and SUMMA-Actors with no batching at 787.08 hours.

Analyzing the results of Experiment 3 further, we see that SUMMA-OpenMP outperforms SUMMA-Actors with batching in terms of job duration by 13.3%. This speedup comes from SUMMA-OpenMP's increased efficiency in computing the GRU physics, where it outperforms SUMMA-Actors with batching by 18.2%. Again, the overhead of using two programming languages hinders the performance of SUMMA-Actors. Nonetheless, SUMMA-Actors with batching still demonstrates effective load balancing, outperforming SUMMA-MPI by 6.1% in job duration. SUMMA-MPI once again performs the best in computing the GRU physics, but it is hindered in overall job duration by the straggler effect.

We can see that SUMMA-Actors with batching performs well overall and is still significantly better than using no batching with a 126.2% difference in job duration. As pointed out in Experiment 1, the `job_actor` faces excessive contention when managing individual `gru_actors`, but this contention is alleviated by the `batch_actors`, as clearly demonstrated in our results.

The first three experiments show that SUMMA-Actors suffers from overhead when computing the GRU physics. This affects the overall performance and allows SUMMA-OpenMP to outperform SUMMA-Actors consistently. We believe that the results would be much closer if SUMMA was written in C++ or if a robust actors library existed for Fortran.

D. Experiment 4: one-month simulation, multi-node scaling

For Experiment 4, we expanded the one-month simulation across 10, 20, and 40 nodes to perform a scaling analysis of SUMMA-Actors with batching, SUMMA-MPI, and SUMMA-MPI+OpenMP. SUMMA-Actors without batching was not included because of its poor performance. The results of this experiment are shown in figure 8.

In each of the three cases, SUMMA-Actors and SUMMA-MPI+OpenMP outperformed SUMMA-MPI. For the 10-, 20-, and 40-node cases, SUMMA-MPI+OpenMP had total job durations of 2.9, 2.14, and 1.64 hours, respectively. SUMMA-Actors with batching had total job durations of 2.97, 2.16, and 1.61 hours, while SUMMA-MPI had total job durations of 4.33, 3.31, and 2.73 hours. An ideal scaling line is included in figure 8 to show the scaling efficiency of each implementation.

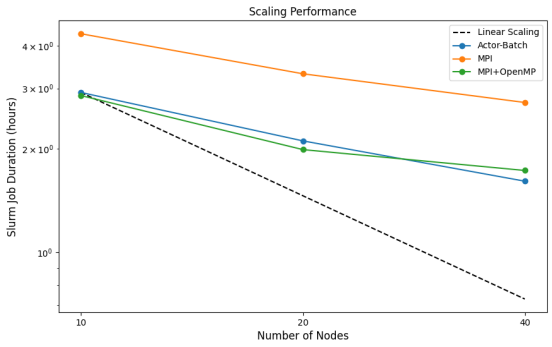

Fig. 8. Scaling analysis of SUMMA-Actors with batching and SUMMA-MPI over 10, 20, and 40 nodes. Each data point was run for a one-month simulation (744 hourly data windows).

It is expected that SUMMA-MPI would perform the worst because it does not balance the workload of GRUs across the CPUs within a node. SUMMA-MPI+OpenMP and SUMMA-Actors have a significant advantage in this regard because they both balance the workload of GRUs across the CPUs within a node. However, the results show that all implementations suffer from the straggler effect between nodes. Linear scaling is not achieved, and this happens because GRU computations are not uniform across data windows, leading to some nodes finishing their data window early and having to wait for all other nodes to finish before moving onto the next data window. This straggler effect is exacerbated when the number of nodes increases, as shown in figure 8.

Comparing SUMMA-MPI+OpenMP and SUMMA-Actors with batching, we see that SUMMA-MPI+OpenMP outperforms SUMMA-Actors with batching in the 10- and 20-node cases. However, in the 40-node case, SUMMA-Actors with batching outperforms SUMMA-MPI+OpenMP. As stated in the previous experiments, SUMMA-Actors is likely hindered by the overhead of using two programming languages. Nonetheless, the disparity in performance between SUMMA-MPI+OpenMP and SUMMA-Actors with batching is not as large as was SUMMA-OpenMP and SUMMA-Actors with batching in the single-node experiments with SUMMA-Actors outperforming SUMMA-MPI+OpenMP in the 40-node case.

The differences in performance between SUMMA-MPI+OpenMP and SUMMA-Actors in the 10- and 20-node cases are 2.3% and 0.9%, respectively. In the 40-node case, SUMMA-Actors outperforms SUMMA-MPI+OpenMP by 7.1%. These results suggest that SUMMA-Actors performs better with smaller numbers of GRUs per node. The highly comparable results show the potential of the actor model in managing the synchronization requirements of highly parallel applications. Even with the overhead of using two programming languages, SUMMA-Actors with batching is able to compete with — and even outperform — SUMMA-MPI+OpenMP in some cases.

V. Conclusions and Future Work

Data synchronization is a common requirement for many problems in scientific and high-performance computing. Two of the most well-known examples where data synchronization is a fundamental feature of the parallelization strategy are the BSP pattern in distributed-memory environments and the fork-join pattern in shared-memory environments. When synchronization is frequent, however, communication overhead can be so high as to cause code performance to suffer.

Using a hydrologic simulation to mimic data assimilation with over 500,000 computational elements, we have described an actor-based batching strategy that buffers communication to mitigate bottlenecks caused by communication overhead. This strategy outperformed the MPI implementation of the same simulation in all single-node and multi-node experiments. However, at the cost of increased overhead due to using two programming languages, SUMMA-Actors was outperformed by SUMMA-OpenMP in all single-node experiments, and in two out of three multi-node experiments for which SUMMA-MPI+OpenMP was used.

Nonetheless, our results demonstrate the effectiveness our actor-based batching strategy in managing the synchronization requirements of highly parallel applications. These results also do not include some powerful features of the actor model that are difficult to measure empirically, such as increased fault tolerance and the decrease in cognitive load associated with the increased ability to reason about the program at a higher level of abstraction.

A powerful feature of the actor model is the implied work-stealing algorithms, which we have demonstrated are highly effective within a node. As part of future work, we plan to investigate dynamic load-balancing strategies that can be applied to reduce the straggler effect in multi-node simulations. We also plan to continue optimizing our actor-based batching strategy, as well as apply it to other algorithms that have high synchronization requirements.

References

[1] A. Pöppl, M. Bader, T. Schwarzer, and M. Glaß, "SWE-X10: Simulating shallow water waves with lazy activation of patches using Actorx10," in *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, 2016, pp. 32–39.

[2] N. Perera, K. Shan, S. Kamburugamuwe, T. A. Kanewela, C. Widanage, A. Sarker, M. Staylor, T. Zhong, V. Abeykoon, and G. Fox, "Supercharging distributed computing environments for high performance data engineering," *arXiv preprint arXiv:2301.07896*, 2023.

[3] R. Hiesgen, M. Nawrocki, A. King, A. Dainotti, T. C. Schmidt, and M. Wählisch, "Spoki: Unveiling a new wave of scanners through a reactive network telescope," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 431–448.

[4] L. Nigro, "Parallel theatre: An actor framework in java for high performance computing," *Simulation Modelling Practice and Theory*, vol. 106, p. 102189, 2021.

[5] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting actor programming in C++," *Computer Languages, Systems & Structures*, vol. 45, pp. 105–131, 2016.

[6] G. Anumba, M. Kiviniemi, M. Chong, and A. Huang, "Digital twins: The convergence of data assimilation and machine learning for predictive energy management in buildings," *Automation in Construction*, vol. 118, p. 103396, December 2020.

[7] J.-C. Loiseau, F. Oquendo, and S. Ducasse, "Data assimilation and digital twins: State of the art and perspectives," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 14–21, March 2020.

[8] S. Jayasuriya, S. Samie, and Y. Zhang, "Data assimilation techniques for digital twin development in predictive maintenance applications," *Procedia CIRP*, vol. 98, pp. 575–580, 2021.

[9] K. Klenk and R. J. Spiteri, "Improving resource utilization and fault tolerance in large simulations via actors," *Cluster Computing*, pp. 1–18, 2024.

[10] M. P. Clark, B. Nijssen, J. D. Lundquist, D. Kavetski, D. E. Rupp, R. A. Woods, J. E. Freer, E. D. Gutmann, A. W. Wood, D. J. Gochis *et al.*, "A unified approach for process-based hydrologic modeling: 2. Model implementation and case studies," *Water resources research*, vol. 51, no. 4, pp. 2515–2542, 2015.

[11] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical computer science*, vol. 410, no. 2-3, pp. 202–220, 2009.

[12] S. Shiina and K. Taura, "Itoyori: Reconciling global address space and global fork-join task parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–15.

[13] T. Yu and M. Pradel, "SyncProf: Detecting, localizing, and optimizing synchronization bottlenecks," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. Association for Computing Machinery, 2016, pp. 389–400.

[14] J. Jahić, K. Ali, M. Chatrangoon, and N. Jahani, "(Dis)advantages of lock-free synchronization mechanisms for multicore embedded systems," in *Workshop Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP Workshops '19. Association for Computing Machinery, 2019, pp. 1–8.

[15] A. Kanade, "Chapter seven - event-based concurrency: Applications, abstractions, and analyses," in *Advances in Computers*, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 379–412.

[16] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, ser. PASCO '07. Association for Computing Machinery, 2007, pp. 24–32.

[17] S. R. Paul, A. Hayashi, K. Chen, Y. Elmougy, and V. Sarkar, "A fine-grained asynchronous bulk synchronous parallelism model for PGAS applications," *Journal of Computational Science*, vol. 69, p. 102014, 2023.

[18] F. M. Maley and J. G. DeVinney, "Conveyors for streaming many-to-many communication," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, Denver, CO, USA, 2019, pp. 1–8.

[19] A. Pöppl, S. Baden, and M. Bader, "A UPC++ actor library and its evaluation on a shallow water proxy application," in *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, Denver, CO, USA, 2019, pp. 11–24.

[20] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[21] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[22] Y. Budanaz, M. Wille, and M. Bader, "Asynchronous workload balancing through persistent work-stealing and offloading for a distributed actor model library," in *2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*, 2022, pp. 39–51.

[23] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279–301, 1989.

[24] L. V. Kalé and A. Bhatele, Eds., *Parallel Science and Engineering Applications - The Charm++ Approach*, ser. Series in Computational Physics. CRC Press, 2013.

[25] V. Freitas, L. L. Pilla, A. de L. Santana, M. Castro, and J. Cohen, "PackStealLB: A scalable distributed load balancer based on work stealing and workload discretization," *Journal of Parallel and Distributed Computing*, vol. 150, pp. 34–45, 2021.

[26] H. Parikh, V. Deodhar, A. Gavrilovska, and S. Pande, "Distributed work stealing at scale via matchmaking," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 250–260.

[27] C. Hewitt, "Actor model of computation: Scalable robust information systems," *arXiv:1008.1459*, 2010. [Online]. Available: https://arxiv.org/abs/1008.1459

[28] J. De Koster, T. Van Cutsem, and T. D'Hondt, "Domains: Safe sharing among actors," in *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, 2012, pp. 11–22.

[29] J. De Koster, T. Van Cutsem, and W. De Meuter, "43 years of actors: a taxonomy of actor models and their key properties," in *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2016, pp. 31–40.

[30] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[31] S. Clebsch, J. Franco, S. Drossopoulou, A. M. Yang, T. Wrigstad, and J. Vitek, "Orca: GC and type system co-design for actor languages," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[32] M. N. A. Khan, "React++: A lightweight actor framework in C++," Master's thesis, University of Waterloo, 2020.

[33] J. Yang and Q. He, "Scheduling parallel computations by work stealing: A survey," *International Journal of Parallel Programming*, vol. 46, pp. 173–197, 2018.

[34] M. P. Clark, B. Nijssen, J. D. Lundquist, D. Kavetski, D. E. Rupp, R. A. Woods, J. E. Freer, E. D. Gutmann, A. W. Wood, L. D. Brekke *et al.*, "A unified approach for process-based hydrologic modeling: 1. Modeling concept," *Water Resources Research*, vol. 51, no. 4, pp. 2498–2514, 2015.

[35] K. Klenk, M. M. Moayeri, and R. J. Spiteri, "High-throughput scientific computation with heterogeneous clusters: A kitchen-sink approach using the actor model," in *Proceedings of the 2024 SIAM Conference on Parallel Processing for Scientific Computing (PP)*. SIAM, 2024, pp. 78–89.

[36] W. J. M. Knoben, M. P. Clark, J. Bales, A. Bennett, S. Gharari, C. B. Marsh, B. Nijssen, A. Pietroniro, R. J. Spiteri, G. Tang, D. G. Tarboton, and A. W. Wood, "Community workflows to advance reproducibility in hydrologic modeling: Separating model-agnostic and model-specific configuration steps in applications of large-domain hydrologic models," *Water Resources Research*, vol. 58, no. 11, p. e2021WR031753, 2022.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 Demonstrates a new application of the actor model to solve synchronization bottlenecks in software.
- C_2 Demonstrates performance improvements from applying an actor-based batching strategy to problems that rely on bulk-synchronous processing or the fork-join pattern in the context of data assimilation applied to a large hydrologic model.
- C_3 Offers a detailed analysis of the performance of the actor model against MPI, OpenMP, and MPI+OpenMP implementations of the same application.

B. Computational Artifacts

All artifacts and their git repositories are listed below. The first five artifacts have specific release versions for this paper at: https://doi.org/10.5281/zenodo.13137302. The last artifact is the forcing data used in the simulations.

- A_1 SUMMA-Actors: https://github.com/uofs-simlab/Summa-Actors
- A_2 SUMMA-MPI: https://github.com/junwei-guo/summa-mpi
- A_3 SUMMA-OpenMP: https://github.com/KyleKlenk/summa/tree/OpenMP
- A_4 SUMMA-MPI+OpenMP: https://github.com/KyleKlenk/summa/tree/MPI-OpenMP
- A_5 Slurm submission scripts and SUMMA settings files: https://git.cs.usask.ca/numerical_simulations_lab/actors/paper_data/data-assimilation-paper
- A_6 Forcing data: https://zenodo.org/doi/10.5281/zenodo.13687421

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figures 1–4
A_1, A_2, A_3, A_5, A_6	C_2, C_3	Figures 5–7
A_1, A_2, A_4, A_5, A_6	C_2, C_3	Figure 8

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

The primary computational artifact supporting all the contributions of this paper is the source code for the version of SUMMA-Actors used in the simulations. This code specifically contains the actor implementations discussed in the paper.

Expected Results

When using SUMMA-Actors in data-assimilation mode, we expect to see the `batch_actor` implementation significantly outperform the same mode that does not use it (C_1, C_2). Additionally, we anticipate that SUMMA-Actors in data-assimilation mode with batching will outperform the same mode implemented with MPI on single and multiple nodes. We expect SUMMA-Actors with batching to be competitive against OpenMP and MPI+OpenMP implementations, however, it may not outperform them (C_3).

Expected Reproduction Time (in Minutes)

The time required to set up this artifact includes installing the necessary dependencies and compiling the code, which we estimate to take approximately 15-30 minutes. The execution time to produce similar results will vary depending on the hardware used, the domain of the simulation, and the length of the simulation.

Artifact Setup (incl. Inputs)

Hardware: This artifact can be run on any modern CPU capable of compiling and running C++ and Fortran code. To reproduce our results for large continental domains, the CPU should be paired with at least 225 GB of RAM and requires a network connection to run in distributed mode. Smaller domains require significantly less RAM.

Software: The software dependencies for this artifact are as follows:

- CMake (3.27.7) – https://cmake.org/
- g++ (12.3.1) – https://gcc.gnu.org/
- Intel(R) Threading Building Blocks (2021.10.0) – https://github.com/oneapi-src/oneTBB
- CAF (1.0.0) – https://www.actor-framework.org/
- Sundials (7.0.0) – https://computing.llnl.gov/projects/sundials
- Netcdf (4.9.2) – https://www.unidata.ucar.edu/software/netcdf/
- SUMMA (PAW-ATM2024 Release) – https://github.com/ashleymedin/summa
- SUMMA-Actors (PAW-ATM2024 Release) – https://github.com/uofs-simlab/Summa-Actors

Datasets / Inputs: The datasets required as initial conditions for SUMMA-Actors are introduced as part of A_5. The forcing data were produced for the continent of North America using the CWARHM workflow https://github.com/CH-Earth/CWARHM. The forcing data is made available as part of A_6.

Installation and Deployment: The steps for compiling and running SUMMA-Actors are included in its repository. The basic steps involve cloning the repository, installing the dependencies, and running the provided CMake script. For installation on a cluster that is part of the Digital Research Alliance of Canada (https://alliancecan.ca/en), like the one used to generate the results in this paper, the specific module

load commands are included in the corresponding compilation script.

Artifact Execution

SUMMA-Actors is run like many other C++ binaries, with various options specified through a combination of command-line arguments and a configuration JSON file. The main tasks include configuring the simulation and executing the simulation. Specific instructions for running the code are included in the README file in the SUMMA-Actors repository. For the files that make up the configuration of the simulation, please refer A_5.

Artifact Analysis (incl. Outputs)

B. Computational Artifact A_2

Relation To Contributions

This artifact was used for comparison against A_1, A_3 and A_4.

Expected Results

The expected results from this artifact are similar to those explained in A_1.

Expected Reproduction Time (in Minutes)

The expected time to reproduce the results from this artifact is similar to the explanation provided for A_1.

Artifact Setup (incl. Inputs)

Hardware: This artifact can be run on any modern CPU capable of compiling and running MPI fortran code. To reproduce our results for large continental domains, the CPU should be paired with at least 100 GB of RAM and requires a network connection to run in distributed mode.

Software:

- CMake (3.27.7) – https://cmake.org/
- gfortran (12.3.1) – https://gcc.gnu.org/
- Sundials (7.0.0) – https://computing.llnl.gov/projects/sundials
- Netcdf-Fortran (4.6.1) – https://www.unidata.ucar.edu/software/netcdf/
- Open MPI (4.1.5) – https://www.open-mpi.org/

Datasets / Inputs: The datasets and inputs for this artifact are the same as those explained for A_1.

Installation and Deployment: The steps for installation are included in the repository for SUMMA-MPI.

Artifact Execution

SUMMA-MPI is run like many other MPI programs. The specific configuration files are explained as part of A_5.

Artifact Analysis (incl. Outputs)

C. Computational Artifact A_3

Relation To Contributions

This artifact was used for comparison against A_1 and A_2.

Expected Results

The expected results from this artifact are similar to those explained in A_1.

Expected Reproduction Time (in Minutes)

The expected time to reproduce the results from this artifact is similar to the explanation provided for A_1.

Artifact Setup (incl. Inputs)

Hardware: This artifact can be run on any modern CPU capable of compiling and running OpenMP fortran code. To reproduce our results for large continental domains, the CPU should be paired with at least 100 GB of RAM.

Software:

- CMake (3.27.7) – https://cmake.org/
- gfortran (12.3.1) – https://gcc.gnu.org/
- Sundials (7.0.0) – https://computing.llnl.gov/projects/sundials
- Netcdf-Fortran (4.6.1) – https://www.unidata.ucar.edu/software/netcdf/
- OpenMP (4.1.5) – https://www.openmp.org/

Datasets / Inputs: The datasets and inputs for this artifact are the same as those explained for A_1.

Installation and Deployment: The steps for installation are included in the SUMMA-OpenMP repository. To explicitly enable OpenMP, ensure that it is linked during the compilation process.

Artifact Execution

SUMMA-OpenMP is run exactly as SUMMA is normally. The specific configuration files are explained as part of A_5.

Artifact Analysis (incl. Outputs)

D. Computational Artifact A_4

Relation To Contributions

This artifact was used for comparison against A_1 and A_2 using multiple nodes.

Expected Results

The expected results from this artifact are similar to those explained in A_1.

Expected Reproduction Time (in Minutes)

The expected time to reproduce the results from this artifact is similar to the explanation provided for A_1.

Artifact Setup (incl. Inputs)

Hardware: This artifact can be run on any modern CPU capable of compiling and running OpenMP fortran code. To reproduce our results for large continental domains, the CPU should be paired with at least 100 GB of RAM.

Software:

- CMake (3.27.7) – https://cmake.org/
- gfortran (12.3.1) – https://gcc.gnu.org/
- Sundials (7.0.0) – https://computing.llnl.gov/projects/sundials
- Netcdf-Fortran (4.6.1) – https://www.unidata.ucar.edu/software/netcdf/
- OpenMP (4.1.5) – https://www.openmp.org/
- Open MPI (4.1.5) – https://www.open-mpi.org/

Datasets / Inputs: The datasets and inputs for this artifact are the same as those explained for A_1.

Installation and Deployment: The steps for installation are included in the SUMMA-MPI+OpenMP repository. To explicitly enable OpenMP, ensure that it is linked during the compilation process.

Artifact Execution

SUMMA-MPI+OpenMP is run exactly as SUMMA is normally. The specific configuration files are explained as part of A_5.

Artifact Analysis (incl. Outputs)

E. Computational Artifact A_5

Relation To Contributions

This artifact is a collection of Slurm submission scripts for the three aforementioned artifacts, along with SUMMA settings files used to generate the results in the paper. The scripts are used to submit jobs to the `Copernicus` and `Niagara` clusters and are organized into specific subdirectories for each cluster. This artifact also contains settings files, excluding the forcing data, in the sundials_settings subdirectory. Each submission script is accompanied by a specific fileManager.txt file that points to the settings files. The README.md file provides instructions on how to run the scripts and specific compilation instructions for the above artifacts for convenience.

Expected Results

The Slurm scripts and settings files included in this artifact can be used to reproduce the results presented in the paper. Additionally, they can be modified to test alternate configurations in corresponding experiments. These files provide a basis for running SUMMA-Actors in data-assimilation mode on a cluster managed by Slurm.

The settings and scripts substantiate the main contributions by providing the means to reproduce the results in the paper. The forcing data is made available as part of A_6.

Expected Reproduction Time (in Minutes)

The setup of this artifact should take 5–10 minutes to clone the repository and make the necessary adjustments to the Slurm scripts and settings files to accommodate the user's system. Execution time will vary depending on the hardware used.

Artifact Setup (incl. Inputs)

Hardware: This artifact can be used to set up and run the above artifacts on any CPU cluster with a Slurm scheduler that meets the hardware requirements outlined for each artifact.

Software: The software dependencies for this artifact are as follows:

- SLURM (22.05.11) https://slurm.schedmd.com/
- LMOD: Environmental Modules System https://tacc.utexas.edu/research/tacc-research/lmod/

Datasets / Inputs: These scripts and settings files require only the addition of forcing data which is made available as part of A_6.

Installation and Deployment: The scripts will require modification to run on different clusters. Most notably, the user will need to change the paths to each executable, the settings files, and the Slurm account to accommodate their system.

Artifact Execution

Running the scripts after the proper adjustments have been made can simply be done with the Slurm command, `sbatch` followed by the path to the script.

F. Computational Artifact A_6

Relation To Contributions

This artifact is the forcing data that was used in all simulations presented in the paper. The forcing data was produced for the continent of North America using the CWARHM workflow https://github.com/CH-Earth/CWARHM.

Expected Results

We expect our results to be reproducible using this forcing data. The forcing data is used as input to the SUMMA-Actors, SUMMA-MPI, SUMMA-OpenMP, and SUMMA-MPI+OpenMP simulations.

Expected Reproduction Time (in Minutes)

To reproduce the forcing data, considerable time is required to run the CWARHM workflow. We estimate this to take approximately 3 to 6 hours, depending on the user's familiarity with the workflow. The longest portion of this process involves downloading the necessary data and running the scripts to process it. Most of this time is spent waiting for downloads or for the scripts to finish running.

Artifact Setup (incl. Inputs)

Hardware: This artifact can be reproduced on any modern laptop, workstation, or cluster with sufficient storage, and an internet connection to download the necessary data.

Software: The software dependencies for this artifact are as follows:

- CHWARM: https://github.com/CH-Earth/CWARHM

Datasets / Inputs: The forcing data does not depend on any other datasets.

Installation and Deployment: The forcing data is generated by running the CWARHM workflow. We refer the specific instructions for running the CWARHM workflow to the CHWARM repository.