Manala: a Flexible Flow Control Library for Asynchronous Task Communication

Matthieu Dreher  
Argonne National Laboratory  
9700 S. Cass Ave.  
Lemont, IL 60439, USA  
mdreher@anl.gov

Kiran Sasikumar  
Argonne National Laboratory  
9700 S. Cass Ave.  
Lemont, IL 60439, USA  
ksasikumar@anl.gov

Subramanian Sankaranarayanan  
Argonne National Laboratory  
9700 S. Cass Ave.  
Lemont, IL 60439, USA  
ssankaranarayanan@anl.gov

Tom Peterka  
Argonne National Laboratory  
9700 S. Cass Ave.  
Lemont, IL 60439, USA  
tpeterka@mcs.anl.gov

Abstract—Tasks coupled in an in situ workflow may not process data at the same speed, potentially causing overflows in the communication channel between them. To prevent this problem, software infrastructures for in situ workflows usually impose a strict FIFO policy that has the side-effect of slowing down faster tasks to the speed of the slower ones. This may not be the desired behavior; for example, a scientist may prefer to drop older data in the communication channel in order to visualize the latest snapshot of a simulation. In this paper, we present Manala, a flexible flow control library designed to manage the flow of messages between a producer and a consumer in an in situ workflow. Manala intercepts messages from the producer, stores them, and selects the message to forward to the consumer depending on the flow control policy. The library is designed to ease the creation of new flow control policies and buffering mechanisms. We demonstrate with three examples how changing the flow control policy between tasks can influence the performance and results of scientific workflows. The first example focuses on materials science with LAMMPS and a synthetic diffraction analysis code. The second example is an interactive visualization scenario with Gromacs as the producer and Damaris/Viz as consumer. Our third example studies different strategies to perform an asynchronous checkpoint with Gromacs.

I. INTRODUCTION

The scientific discovery process is composed of parallel tasks exchanging data. Tasks can form a graph where a node is a parallel task and an edge is a communication channel between two parallel tasks. Traditionally, parallel tasks exchange data through files. However, the growing gap between computation capabilities and available I/O bandwidth on current and future supercomputers [1], [2] motivates in situ analysis. In this context, data are exchanged while being produced in memory and are sent directly to the consumer, avoiding the I/O bottleneck.

Several in situ infrastructures [3], [4], [5], [6] adopt a space-partitioning execution mode: each task runs independently on dedicated resources and exchanges data asynchronously. However, tasks may not process data at the same speed, potentially causing overflows in the communication channel between them. In order to prevent this problem, software infrastructures for in situ workflows usually control the flow of messages with a FIFO policy that has the side-effect of slowing faster tasks to the speed of the slower ones.

We posit that applications ought to decide how data are exchanged between components, not in situ infrastructures. An in situ workflow is the translation of a scientific process into a graph of tasks exchanging data. Scientists should determine the flow control requirements between those tasks. For example, in interactive applications such as in situ visualization or simulation steering, the focus is to visualize the current state of the simulation. FIFO may not be the best policy when the visualization processes data at a different pace from that of the simulation. For this case, it is preferable to send only the most recent frame to the visualization when the renderer is ready to process a new iteration. However, in situ infrastructures [3], [7], [4], [8], [9] manage the flow of messages between tasks, leaving little flexibility to the user.

To address these challenges, we present Manala, a flow control library that enables the user to control the data exchange between parallel producers and consumers. Manala creates on top of existing in situ middleware a thin layer managing the message queue between a producer and consumer. Manala intercepts messages from the consumer, stores them, and selects the messages to send to the consumer based on a flow control policy. Manala provides an extensible hierarchy of memory layers ranging from the main memory to the file system. This allows Manala to buffer as many messages as possible before blocking the producer. The library gives access to the message queue allowing the user to modify the order of messages or drop messages. This mechanism enables the user to create complex flow control policies to meet application requirements.

We demonstrate the capabilities of Manala with three different applications. First, we present a pipeline analysis of data produced by LAMMPS [10], a molecular dynamics (MD) simulation. All frames produced by the simulation must be analyzed. However, the analysis processes data slower than the simulation, which requires data to be buffered. The second example demonstrates a case of interactive in situ visualization with Gromacs [11], another MD simulation, and Damaris/Viz [12], a framework to connect in situ an application to VisIt. We send only the most recent frame to the visualization. The third example applies a hybrid regular and best effort sampling strategy to adaptively write data from Gromacs.
Manala manages communication between parallel tasks. We present in this section relevant work on in situ infrastructures and focus on how they manage asynchronous task communication.

A. Communication Libraries

Several low-level protocols exchange data between two points. The Transmission Control Protocol (TCP) is a simple interface widely supported by many network interfaces. More specific network hardware, such as that found in supercomputers, require specific optimization. Hardware vendors often provide a low level API for connecting and exchanging data between two points. Examples include OpenFabrics Verbs, Myrinet Express, and Portals. CCI [13] implements a common interface on top of these APIs.

The Message Passing Interface (MPI) [14] is the base communication layer for many parallel applications such as simulation [11], [10] and analysis codes [15] but also in situ infrastructures [8], [7], [4]. MPI defines synchronous and asynchronous point-to-point and collective communication operations.

Mercury [16] is a C library for implementing remote procedure calls, optimized for high-performance computing systems. Mercury has support for remote function calls that include large data arguments. It relies on CCI to manage its communications.

ZeroMQ [17] is a client/server library. It allows several clients to connect to the same server port and takes care of message transfer, buffering, and aggregation in the background. It relies on TCP sockets for communication. ZeroMQ buffers messages internally. If the buffer size reaches a threshold, the producer is blocked. Nanomsg [18] improves the performance of ZeroMQ by using the native vendor API instead of TCP sockets.

All the solutions presented in this section implement a point-to-point FIFO communication channel between two peers. However, they can be used as a base layer to implement more complex communication patterns.

Bredala [19] is a library to redistribute data between two parallel programs. The library protects the semantics of the data during the redistribution of data. It provides several redistribution patterns to match application needs. Bredala provide a FIFO communication channel between a parallel producer and consumer based on MPI.

B. In Situ Infrastructures

In situ infrastructures exchange data between parallel tasks. The execution mode of the tasks determines the scheduling of the tasks and the communication between them.

In time partitioning mode, the producer and consumer run in the same processes and are executed sequentially. Every time a producer sends data, the consumer is called and processes the data available. This mode involves no risk of overflow because the consumer is executed synchronously while the producer is blocked. In situ visualization tools such as Catalyst [8] and Libsim [7] adopt this mode.

In space partitioning mode, the producer and consumer run asynchronously on different resources. This approach reduces the impact on the producer performance because it is nonblocking.

Damaris [4] splits the MPI communicator of the producer to reserve dedicated cores or nodes to execute the consumer. Messages are allocated in a shared memory buffer. If one of the producer processes fails to allocate a buffer, for instance because of overflows, the data associated with the current iteration will not be processed, and the data will be lost.

Dataspaces [20] is a distributed in memory key/value store for scientific applications. Data are stored in dedicated processes usually hosted on dedicated nodes. Dataspaces provides a publisher/subscriber mechanism for applications to store and retrieve data in the system. Applications send data asynchronously to Dataspaces via Dart servers [21]. The user is responsible for removing data on the server once the data are no longer necessary. Consequently, overflows can arise if the consumer does not process and remove data fast enough.

ADIOS [22] is a common interface for several in situ infrastructures [3], [23], [24]. Data are described in a separate XML file with a name, a type, and a layout. Data can be organized by groups with a transport method for each group. The transport method can be an I/O method such as MPI-IO or an in situ system. Some of these systems enable communications from $N$ to $M$ processes. Several in situ transport methods are built on top of EVPath [25]. EVPath queues messages on the producer side and sends data upon request from the consumer. If the queue is full, the runtime will block the producer.

Decaf [5] is a dataflow system for the parallel communication of coupled tasks in an HPC workflow. The dataflow, which is a connection between a producer and a consumer, is enhanced with computational resources to transform incoming data and perform complex redistribution.

All these infrastructures adopt a strict FIFO policy to exchange data between tasks. Manala can be used on top of infrastructures such as EVPath to apply user-defined flow control polices on the communication between a producer and a consumer.

FlowVR [6] is a middleware to create in situ workflows. The graph is described through a Python API. FlowVR supports FIFO point-to-point communications, but the Python API eases the creation of more complex communication patterns. FlowVR provides a series of components called Filters and Synchronizers. Filters can buffer data between a producer and consumer. Synchronizers manage the frame counts and decide which frame to send to the consumer upon request. Manala adopts a similar approach to manage flow control, separating the buffering mechanism from the message queue management, but with several improvements. First, our components can be executed entirely asynchronously from the tasks. Second, we provide an extensible list of possible
storage. Third, communications are done in parallel. Fourth, our approach is compatible with supercomputers platforms.

C. Time Constraints and Flow Control

Another way to control the flow of data is to constrain the execution time of the consumer. Malakar et al. [26] proposed an analytical model to schedule several analyses (consumers) given a time constraint. The user provides a time budget during which the system can run analysis. For each analysis, the user provides an execution time and priority level. The model then predicts an optimal scheduling for analysis executed synchronously in time-partitioning mode. This work has been extended in [27] for space partitioning mode. Running the analysis on dedicated resources creates new constraints. In this work, the linear model takes into account time and memory requirements of the analysis, their importance, minimum interval, maximum available storage space and memory per node, and the simulation rate.

Dorier et al. [28] adapts the input data of an in situ visualization to meet a performance constraint. Their framework evaluates the variability of each block of data to visualize. The goal is to simplify blocks of data representing little information in order to improve rendering performance. The runtime monitors its performance and dynamically adapts its filtering level to meet the simulation’s run-time constraints.

III. APPLICATION DRIVERS

In this section, we show three applications and describe why a strict FIFO flow control policy is not satisfactory for their needs.

A. Diffraction Pattern Analysis

In materials science, the structural response of single nanoparticles at femtosecond and picosecond timescales is not well understood. Investigating the temporal behavior of such nanomaterials under conditions of external thermal stimulation is crucially important for energy research. However current experimental methods such as Bragg coherent diffraction imaging (BCDI) have a resolution limit of 10 nm. Molecular dynamics simulation codes such as LAMMPS [10] are used to provide atomic-level resolution to compare with and augment the experimental results.

Scientists use Debyer [29] to analyze data produced by LAMMPS and compute synthetic diffraction patterns to compare with experimental ones. For this application, scientists must analyze each data output produced by the simulation. Computing the diffraction patterns is expensive because Debyer performs computation on each pair of atoms ($O(n^2)$). In comparison, LAMMPS’ complexity for our use case is linear in the number of atoms. Consequently, the analysis requires more time than the simulation does to complete.

That application scenario is not ideal for traditional FIFO flow control policies because the consumer (Debyer) is processing data slower than the producer (LAMMPS). In this case, in situ infrastructure would slow LAMMPS to the pace of Debyer, wasting computational resources.

B. Interactive In Situ Visualization

In situ visualization allows users to obtain information about a running simulation. For instance in biology, in situ visualization is used to interactively steer a molecular dynamics simulation [30] such as Gromacs to reach certain configurations.

Gromacs can run at several hundreds of iterations per second, an order of magnitude faster than common atomistic renderers. Consequently, a FIFO flow control policy will slow Gromacs because of a slower renderer. However the user does not need to see each iteration produced by Gromacs in order to steer the simulation. The user needs to see only the current state of the simulation in order to guide the simulation. That approach is not possible with a FIFO flow control policy.

C. Adaptive Analysis Output Writer

Simulations such as Gromacs periodically write a snapshot of the simulation state to disk. Writing analysis output to disk with Gromacs is costly because it is done synchronously during the simulation. Choosing the output frequency is critical for the simulation performance. A high output frequency increases the temporal resolution of the output but simulation execution blocks more frequently. A lower output frequency is less costly but results in potentially missing important events between two outputs.

Writing the snapshot in space partitioning reduces the impact on the simulation performance and allows writing data more frequently [6]. Yet, with a FIFO flow control policy, the writer can still block the simulation if the Gromacs output frequency is too high or if the writer bandwidth slows because of jitter in the file system. A more dynamic approach is desired.

IV. MANALA ARCHITECTURE

Manala is a library designed to create flexible flow control policies for in situ workflows. Manala builds on top of traditional in situ middleware using FIFO communication channel infrastructure to buffer and manipulate the flow of messages exchanged between a parallel producer and consumer. Manala comes with an extensible interface allowing a developer to create new data exchange policies and buffering capabilities. We first present the general architecture of Manala before describing in more detail specific features.

A. Expanding Communication Channels

Manala is designed to control a flow of messages between a producer and consumer (Figure 1a). It is meant to be integrated on top of preexisting in situ infrastructures. Manala requires the creation by in situ infrastructure of an intermediate task between the producer and the consumer in space division mode (Figure 1b). We call the intermediate task a link. Manala assumes that the communication channels between the tasks are FIFO. Infrastructures such as EVPath, FlowVR, and Decaf meet these requirements and are possible targets for Manala. Manala repurposes the link to manage the flow of messages between the producer and consumer. The link
Fig. 1: (a) Basic pipeline with a producer and a consumer. (b) The in situ infrastructure inserts an intermediate task between the producer and the consumer called link. (c) The Manala library build an infrastructure on top of the three-task pipeline to control the flow of messages between the producer and consumer.

intercepts messages from the producer, stores them, and selects messages to forward to the consumer upon request (Figure 1c). The objective of this organization is threefold: (1) preventing overflows on the producer and consumer side, (2) avoiding blocking the producer as much as possible, (3) providing a mechanism to the user to manage the message queue between two tasks.

This organization presents two advantages. First, it decouples the queue management execution from the producer and consumer execution flow. This approach allows the link to receive producer’s messages while answering consumer’s queries at the same time. Second, the link possesses its own computational and memory resources. The processing of the message queue is done asynchronously, limiting the impact on the producer and consumer performance. It also allows us to separate the memory needs of the user application from the queue memory. This feature is particularly useful for memory-bound applications, for instance. In that case, the producer simply sends its data to the link, which can allocate memory on dedicated nodes to buffer the data.

B. Storage Hierarchy

Every time the link receives a message from the producer, it inserts the data into a storage collection. When initializing a Manala communication channel, the library creates an empty storage collection. The user then attaches storage layers to the collection.

A storage layer corresponds to a storage method that implements a mechanism to store and load data identified by a unique token. Currently, Manala provides three storage methods: main memory, shared remote memory, and files. The main memory layer keeps the data in local memory. The shared remote memory layer serializes data and sends the data to remote servers. Our implementation relies on a modified implementation of Dataspaces servers [20], a distributed in memory key/value store, to store and load data. The file storage method serializes data and writes the data into a binary file.

When the link inserts data inside the storage collection, the collection tries to insert the data into one of the attached layers and return a token identifying the data. That token can be used later to retrieve or erase the associated data.

By default, the storage collection adopts a greedy strategy trying to insert data in the first available layer. Storage layers are usually attached by speed: the first layer is the fastest available, and the last layer is the slowest. The developer can extend the storage collection interface to implement other strategies. For instance, for interactive application, the consumer requires the most up-to-date data available. Consequently, a more efficient solution is to always insert new data into the fastest layer available and move older data to a slower layer or erase data. We will show an example of this strategy in Section III-B.

The storage layers can be extended to support new memory hardware and various buffering libraries or to implement a different storage policy on preexisting storage layers. For instance, the current main memory layer is implemented as a map for portability reasons. This allows the layer to access any data at any time. Other structures are more efficient for certain access patterns. For instance, a queue is more efficient for a FIFO access pattern. The same interface can also be used to support new hardware. For instance, future supercomputers will be equipped with NVRAM accessed by the file system or a specific API. Our generic interface allows developers to easily integrate these new hardware into Manala.

C. Selector Interface

The selector implements a flow control policy, that is, selecting which message the producer sends and which message the consumer receives. Manala provides an abstract interface to implement new flow control policies. Currently Manala has four selectors: all, some, latest, and hybrid. The all selector implements a traditional FIFO flow control policy. The consumer receives messages in the same order as they were received by the link, with no messages dropped. The some selector stores one message out of every \( x \) messages received and drops the others, with \( x \) a user-defined parameter. The stored messages are then sent to the consumer in the same order as they were received. The latest selector stores every message received. When the link receives a request from the consumer, the selector sends the most recent data received and drops older data. The hybrid selector is a combination of the some and latest selectors. The selector stores one message out of every \( x \) message received and send the most recent data received upon request.
The producer and link processes host a **Selector** object. A selector is responsible for deciding (1) whether a producer should send or drop a message, (2) whether a message should be stored or not by the link, and (3) which data to send to the consumer upon request.

Figure 2 presents a flowchart of the selector. When the producer tries to send a message, the selector decides whether the message should be sent or dropped based on the flow control policy. This filters messages that will never be sent to the consumer. For instance, the *some* selector drops \((x - 1)\) messages out of \(x\), before any communication over the network. When the link receives a message from the producer, the selector decides whether the message should be stored or dropped. If the message is stored, the selector saves the message ID internally. When the link receives a data request from the consumer, the selector selects which data to send and how to update the storage collection, based on the flow control policy and the saved message IDs. The selector returns a command for the storage collection to retrieve the selected message and update the storage collection. Updating the storage collection can include erasing the retrieved data and possibly older data as well. For instance, a *latest* selector request will retrieve the latest message received and erase the entire content of the storage collection.

For some policies, the selectors on the link cannot decide independently which data to send to the consumer because they do not have the same content in their storage collection at the same time. That is, for instance, the case of the *latest* selector. Consequently, there is no guarantee that all the selectors would select the same message to forward leading to incoherent messages. Instead, the selection process is centralized: only one selector decides which message to send and broadcasts the selection to the other selectors hosted by the links.

### D. Communication Protocol

Manala is responsible for coordinating the data exchanges between the producer, consumer, and link. For that purpose, Manala sends commands to (1) block or unblock the producer if the storage collection becomes full or available, (2) notify the link that the consumer is ready, and (3) broadcast the selection command to all links. Manala cannot use the same communicators as the data message because the communication patterns are not the same. Instead, Manala creates a separate communicator network, presented in Figure 3.

Manala creates a communicator for the producer and the link. This allows the root MPI rank of each program to broadcast commands to their respective processes. Manala also sets up two point-to-point system communicators: one between the root of the consumer and the root of the link and one between the root of the link and the root of the producer. They both form a feedback loop opposite to the data communicators. Each of these communicators serves a different purpose. The communicator between the producer root and the link root sends a blocking or unblocking command to the producer when the storage collection of the link is full.

The communicator between the consumer root and the link root notifies the link that the consumer is ready. The link communicator is used to broadcast the selection command. The producer communicator is used to broadcast the blocking and unblocking command.

Manala provides a class called **Stream** implementing this communication protocol. The stream class is the interface between Manala and the in situ infrastructure. It implements three functions: `processProd()`, `processLink()`, and `processCon()`. These functions are called by the producer, the link, and the consumer, respectively. The stream comes with three variations: `StreamDoubleFeedback`, `StreamSingleFeedback`, and `StreamNoLink`. *StreamDoubleFeedback* implements the protocol as described in this section. *StreamSingleFeedback* implements the link communication channel and the loopback from the consumer to the link. However, the producer does not listen to the link before sending messages. That implementation reduces the latency of the producer for cases where the link can guarantee that it will not run out of memory. We describe an example using this strategy in Section V-B. *StreamNoLink* controls a communication channel without a link (two-task pipeline). The stream creates a loopback from the consumer to the producer blocking the producer until the consumer is ready to receive new data. Section V-A describes a use of this strategy.

### E. Integration with Decaf

We integrated Manala with Decaf [5], an in situ middleware able to generate and execute graphs of tasks. Decaf expends a communication channel between two tasks with an intermediate area. The intermediate area can transform or modify messages sent from the producer to the consumer. Internally, Decaf organizes its communication channels in *dataflows*. A dataflow includes a producer, a consumer, and an intermediate...
Fig. 2: Flowchart of the selector. The selector is run by the producer and the link. When the producer wants to send a message, the selector decides to send or drop the message depending on the flow control policy. On the link, the selector decide to store or drop incoming messages. Upon reception of a notification from the consumer, the selector on the link decides which data to send to the consumer.

```
"nodes": [
  {
    "nprocs": 2,
    "start_proc": 0,
    ...
  }

  {
    "nprocs": 1,
    "start_proc": 4,
    ...
  }

"edges": [
  {
    "nprocs": 2,
    "start_proc": 2,
    "stream": "double",
    "storage_collection_policy": "greedy",
    "storage_types": ["mainmem", "file"],
    "max_storage_sizes": [1, 10],
    "frame_policy": "some",
    "prod_output_freq": 5,
    ...
  }
]
```

Listing 1: JSON sample to generate a communication channel between a producer and a consumer. The link uses a some selector and two layers of storage (main memory and files).

area. The dataflow also provides the data communicators between tasks.

We integrated Manala in two steps. First, we extended the JSON configuration file of Decaf to include the different parameters necessary to configure the streams, storage collection, and selector policy. Listing 1 presents the relevant sections of the configuration file. Second, we integrated the streams inside the dataflow object of Decaf. The dataflow is naturally organized as a three-task pipeline and incorporates a data plane between tasks. Manala retrieves the different communication channels from the dataflow to build its own system communication channel. On the producer side, the function `processProd()` is called during a call to Decaf `put()` from the user. On the consumer side, the function `processCon()` is called during a call to Decaf `get()`. Decaf runs an infinite loop on the intermediate area waiting for incoming messages, transforming the data, and forwarding the messages to the consumer. For each iteration, the intermediate area calls `processLink()`, which selects a message that is then given to Decaf to forward to the consumer.

F. Checkpointing the Link

Applications such as simulations usually provide a checkpoint mechanism against software or hardware failures. Manala stores data in different ways on the links. The location and storage method of the data vary depending on the storage layers selected by the user. Once a producer sends data to the link, there is no guarantee when the consumer will receive it. Depending on the selector used, the data may be stored by the link and sent to the consumer later. In case of failure, the data stored by the link would be lost definitely.

Manala provides a checkpoint mechanism to save the messages stored by the link. Each link process stores its messages into separated files. Each file is identified with the message id and the rank ID of the link saving the message. Once all the messages are stored, each process write a metadata file listing all the files saved by the link. By default, Manala uses the file system storage layer to store messages on disk. If a message is already in the file system storage layer, the message is simply added to the list of saved files. At restart, the links read the metadata files and load back the messages.

The checkpoint can be triggered by the user with a call to the function `save()` or optionally with `terminate()` (disable by default) on the producer side. Manala also provides a simple module reader able to read the metadata files from the links and load back the messages. The reader can act as a producer forwarding the data previously stored by a link. We show in Section V-A a usage example for this functionality.

V. Experiments

In this section, we present three different applications based on materials science and molecular dynamics. Our first example presents an application when the consumer is processing data more slowly than the producer. Our second example is an interactive application. We demonstrate how our latest policy can help a user interact with a simulation while limiting
the impact on the simulation performance. Our third example analyzes different approaches to write data snapshots.

The following tests were conducted on Froggy, a 138-node cluster from the Ciment infrastructure. Each compute node is equipped with two 8-core processors, Sandy Bridge-EP E5-2670 at 2.6 GHz with 64 GB of memory. Nodes are interconnected through an FDR InfiniBand network. For all experiments we used the molecular model of a box of gold composed of 500,000 atoms.

### A. X-Ray Diffraction Pattern Study

Our first example couples a molecular dynamics simulation, LAMMPS [10], to Debyer [29], a postprocessing tool designed to compute diffraction patterns. In this workflow, LAMMPS is the producer and Debyer the consumer. Both tasks are MPI programs. We used the library interface of LAMMPS to extract the atom positions and their respective types and send them to the link. We modified Debyer to read input data from Decaf communicators rather than files. Debyer requires each process to have all the full dataset for its computations. Hence, we extracted all the atom positions from each process of LAMMPS and sent them to the links. The links then broadcast the full dataset to Debyer upon request.

For all scenarios, we extracted atom positions from the simulation every 2,000 iterations.

We first connected LAMMPS and Debyer without links using the stream `StreamNoLink`. In this configuration, LAMMPS will block until Debyer is ready to process a new output data. Figure 4 presents the time decomposition of LAMMPS and Debyer to produce and analyze five simulation outputs. Each task ran on 512 cores (separate sets of 32 nodes each). LAMMPS needs on average 5 seconds to compute 2,000 iterations and output data. Debyer requires on average 84 seconds to analyze one output data. Consequently, LAMMPS was blocked 94% of the time, wasting computational resources.

We introduced a link between LAMMPS and Debyer with buffering capabilities using the `all` selector and the stream `StreamSingleFeedback`. We attached to the storage collection a main memory layer absorbing up to three simulation outputs and a file storage layer. Two nodes (32 cores) were allocated to host the links.

Figure 4: Execution timeline for LAMMPS and Debyer without link (5 iterations). Blue squares are computation times and red square are waiting times. LAMMPS alternates between computation time (4s) and idling time (80s), waiting for Debyer to be ready to process new output data.

Figure 5 presents the buffered time decomposition of LAMMPS and Debyer for the first five iterations. LAMMPS requires between 400 and 600 ms to send its data output toward the link before starting the next step. Since the storage collection has virtually infinite storage because of the file system layer, LAMMPS can produce data at its own pace without being blocked at any time.

LAMMPS completes its task faster than Debyer does in this configuration. As a result, the resources allocated for LAMMPS are idling at the end of the simulation while Debyer completes its task. To avoid wasting CPU hours, we can utilize the checkpointing capability of Manala. First, we run the workflow as described previously. When LAMMPS sends the terminate signal, Manala saves the data stored in its storage collection and terminates Debyer, completing the first job. Second, we launch a new job with only Debyer and the reader module from Manala. We connect the two of them with a stream `StreamNoLink`. This strategy avoids any idle time on both LAMMPS and Debyer while minimizing the number of CPU hours consumed.

A better solution would be to reallocate the resources of LAMMPS to Debyer automatically once LAMMPS has completed its task. Once middleware like Decaf and FlowVR can reconfigure graphs of tasks at runtime, this mechanism will enable reassigning freed computational resources to other tasks.

The code to run these experiments is available on Bitbucket.1

### B. Interactive Molecular Dynamic Visualization

Our second application is the in situ visualization of the FepA protein, a channel in the periplasm of Gram-negative bacteria. Compounds can traverse the membrane of the cell through this channel. In this case, we study the traversal of an iron complex. This mechanism is of interest to biologists because drugs pass through these channels. Yet, the exact path and biological mechanisms allowing the iron complex to go through the channel are unknown. Biologists are interested in visualizing the density of atoms in the FepA channel, because areas of low density are potential paths for the iron complex.

1https://bitbucket.org/matthieu_dreher/lammps_diffraction
A possible method to visualize these areas is the isosurface algorithm available in VisIt.

We connected a molecular dynamics simulation Gromacs [11] to an in situ visualization. The interactive visualization allows biologists to monitor the simulation and observe correlations between the iron complex position in the channel and low-density areas.

We modified Gromacs with the Decaf API to extract atom positions and their respective IDs from the simulation and send them to a visualization task. The visualization task computes a density grid and sends the grid to VisIt through Damaris/Viz. Damaris/Viz embed VisIt servers in the visualization task that will compute the isosurface. However, the servers compute the isosurface only if a VisIt client is connected to the servers. Molecular dynamics simulations are fast-paced codes that process several hundred iterations per second at scale. In comparison, the isosurface algorithm is costly for large molecular models because it requires the computation of a density grid and a marching cube. Consequently, the visualization is processing data slower than the simulation at scale.

The following tests were conducted on Froggy. We used Gromacs 4.5.5 and the FepA model composed of about 70,000 atoms. For all scenarios, we extracted atom positions from the simulation every 100 iterations. The visualization task runs Damaris 1.0.1 with VisIt 2.10.3. For all this section, Gromacs ran on 480 cores (32 nodes, 15 cores per node out of 16), the link ran on 32 cores (the remaining core per node on Gromacs nodes), and the visualization task ran on 4 cores on a dedicated node. We limited the visualization task to four cores because it ran on a visualization node shared with other users.

We first connected the simulation to the visualization task with the *some* selector. We output data from the simulation every 100 iterations. Each link had attached a single main memory layer in its storage collection storing up to 10 messages. Figure 6 presents the execution time of 100 iterations of Gromacs over time. Three phases are observable. The first phase from the iteration 0 to 19,300 corresponds to the time of Gromacs without a VisIt client connected to the visualization task. On average, Gromacs output data every 233 ms while the visualization task requires less than 10 ms to process the data. During this phase, the visualization is processing data fast enough to avoid blocking Gromacs. From iteration 19,300 to 26,999, we connected a VisIt client to the visualization task and rendered the isosurface. On average, the rendering requires 1,338 ms, more than the simulation time. From the iteration 19,300 to 20,700, VisIt is rendering frames, but Gromacs’s speed is not affected. That interval corresponds to the time to fill the buffer. From the iteration 20,700 to 26,999, the buffer is full; Manala has to block the simulation regularly to free space in the storage collection. From the iteration 26,999 to the end of the simulation, we disconnected the VisIt client from the visualization. As during the first phase, the visualization task is fast enough to process simulation data, avoiding blocking the simulation. Overall, we see that even with buffering capabilities, the simulation is blocked eventually because the visualization time is more than five times longer than the simulation time.

For this study, biologists do not actually need to see every output from the simulation. It is more important to visualize the current state of the simulation. We replaced the *some* selector by the *latest* selector and replayed the same experiment. To ensure reproducibility, we recorded the execution time of VisIt for each iteration during the previous experiment and replaced VisIt calls with sleep in the visualization task. This ensures that the visualization tasks takes the same execution times for both runs. Figure 7 shows that the simulation...
<table>
<thead>
<tr>
<th>Setup</th>
<th>Execution Time (s)</th>
<th># Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>gromacs</td>
<td>148</td>
<td>5000</td>
</tr>
<tr>
<td>gromacs-100</td>
<td>276</td>
<td>500</td>
</tr>
<tr>
<td>gromacs-10</td>
<td>1692</td>
<td>5000</td>
</tr>
<tr>
<td>manala-some-100</td>
<td>156</td>
<td>500</td>
</tr>
<tr>
<td>manala-some-10</td>
<td>390</td>
<td>5000</td>
</tr>
<tr>
<td>manala-adapt</td>
<td>209</td>
<td>1500</td>
</tr>
</tbody>
</table>

**TABLE I**: Average total execution time of Gromacs with different writing methods. *Gromacs* gives the baseline performance of Gromacs without I/O. *Gromacs-100* and *gromacs-10* use the native method of Gromacs to output data every 100 and 10 iterations. *Manala-some-100* and *manala-some-10* use Manala and the *some* selector to write data. *Manala-adapt* uses our custom selector with a high frequency of 10 and a low frequency of 100.

performance is not blocked anymore by Manala during the interaction phase. The reason is that Manala forwarded only 12 iterations to the visualization task during the second phase and dropped 49 iterations. With the *some* selector, the visualization task processed 61 iterations. This strategy satisfies the user needs while hiding the visualization processing time from the simulation performance.

The code to run these experiments is available on Bitbucket.²

### C. Adaptative Snapshot Writing

In this section, we study different methods to write a snapshot (or trajectory) with Gromacs as our application driver. We used Gromacs 4.6 with a molecular model of large membranes composed of about 2,100,000 atoms. For all experiments, Gromacs ran on 480 cores (32 nodes, 15 cores per node) during 50,000 iterations. The trajectory included atom positions, speeds, and forces.

We first wrote a trajectory with the native method of Gromacs every 10 (high-frequency) and 100 (low-frequency) iterations. Table I presents the average (out of 5 runs) total execution time of Gromacs without any I/O (gromacs) and writing a trajectory at low (gromacs-100) and high frequency (gromacs-10). Writing a trajectory every 100 iterations increases the simulation time by 86%, while writing every 10 iterations increases simulation time by 1043%. Gromacs writes its trajectory by synchronously gathering all the atom information on the MPI rank 0 to write a single file, significantly degrading the simulation performance. Because of this cost, biologists usually write trajectories at a low frequency.

We modified Gromacs with the Decaf API to extract the atom positions, forces, and speeds. We created a parallel task writing the data received from Gromacs into a file. The writer receives the data in parallel but gathers the data to a single process to write the data. For the following experiments, the writer ran on 16 cores (1 node). We added one core per simulation node (32 cores total) to host the links. Our first setup uses the *some* selector with an output every 100 and 10 iterations. Each link had attached a main memory storage layer able to store up to 10 iterations. *Manala-some-100* and *manala-some-10* give the execution time of Gromacs with data sent every 100 and every 10, respectively, to the writer. Writing a trajectory every 100 iterations increases the simulation time by 5%, while writing every 10 iterations increases the simulation time by 163%. This method considerably reduces the cost of writing a trajectory because the trajectory is written asynchronously on dedicated resources. The remaining cost on the simulation is due to the serialization of Gromacs data into a message and the competition between Gromacs and Decaf to access the network card. At low frequency, the simulation is never blocked by Manala because the writer has enough time to process incoming data. At high frequency, Manala has to block the simulation regularly: the simulation outputs data every 36 ms while the writer needs 53 ms on average to process one iteration.

Although the impact of writing a trajectory is significantly decreased with this strategy, the scientist still has to make a tradeoff to select the output frequency. We developed a custom selector to adapt the writing frequency of the trajectory to the pace of the writer. The selector takes as a parameter a high and low frequency output. The selector tries to forward data to the consumer at the high frequency but guarantees to forward data to the consumer only at the low frequency. For this example, we extracted data from Gromacs every 10 iterations (high frequency). The selector applies an *all* strategy for messages with an id of a multiple 100 and a *latest* strategy for the other messages. That means that the simulation will block only if the writer requires more time than the period of the low frequency. This strategy (*manala-adapt*) increases Gromacs execution time by 41%. This selector reduces the cost of the high frequency output by writing fewer iterations to disk. During *manala-some-10*, the writer saved 5000 iterations to disk while during *manala-adapt* the writer saved 1509 iterations on average. We note that the cost on the simulation execution time is due mainly to the serialization of Decaf. Manala enables developers to implement their own flow control to improve existing flow control policies based on their needs.

The code to run these experiments is available on Bitbucket.³

### VI. Conclusion

We introduced Manala, a flexible flow control library designed to manage the flow of messages between a producer and a consumer in an in situ workflow. Manala intercepts messages from the producer, stores them, and selects the message to forward to the consumer based on a flow control policy. Manala currently implements four flow control policies that we identified with our users as the most relevant for their use cases.

We demonstrated the benefits of moving from a strict FIFO flow control policy to more advances strategies with three different applications driven by our users. We first reduced the number of CPU-hours consumed by an analysis pipeline.

²https://bitbucket.org/matthieu_dreher/gromacs_interactive

³https://bitbucket.org/matthieu_dreher/gromacs_checkpoint
using Manala buffering capabilities. Second, we hid the cost of in situ visualization during an interactive session. Third, we explored several methods to write a snapshot for future analysis.

Future supercomputer platforms will incorporate new types of memory hardware such as burst buffers or NVRAMs. We will develop new storage layers taking advantage of these new hardware to buffer data. We will also work on parallelizing the links of Manala to improve their receiving and sending response time.

Manala is part of the Decaf project and is available in open source.4

ACKNOWLEDGMENT

This material was based upon work supported by the U.S. Department of Energy, the Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, program manager Lucy Nowell.

Some of the computations presented in this paper were performed on the Froggy platform of the CIMENT infrastructure (https://ciment.ujf-grenoble.fr), supported by the Rhône-Alpes region (GRANT CPER07_13 CIRA) and the Equip@Meso program (reference ANR-10-EQPX-29-01) of the programme Investissements d’Avenir supervised by the ANR.

REFERENCES


The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (Argonne). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan.