FH Aachen University of Applied Sciences
Campus Jülich

Department of Medical Engineering and
Technomathematics

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

In Transit Coupling of Neuroscientific Simulation and Analysis on High Performance Computing Systems

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in Technomathematics

Kim Sontheimer

Jülich, August 26, 2019

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Masterarbeit mit dem Thema **In Transit Coupling of Neuroscientific Simulation and Analysis on High Performance Computing Systems** selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Ich verpflichte mich, ein Exemplar der Masterarbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Ort und Datum

Unterschrift

Diese Arbeit wurde betreut von:

Prof. Dr. rer. nat. Stephan Bialonski
M.Sc. Wouter Klijn

Diese Arbeit wurde erstellt am
Jülich Supercomputing Centre

Abstract

High performance computing (HPC) is experiencing an increasing imbalance between processing power and I/O capabilities. This imbalance has led to the challenge of managing the large amounts of data produced by extreme-scale simulations. It has become prohibitive expensive to store this data on disk for subsequent offline analysis. *In transit* processing could perform this analysis on memory-resident data.

In this thesis, based on requirements of neuroscientific use cases, a framework has been designed, implemented and tested on the JURECA supercomputer located at the Forschungszentrum Jülich. In the framework, simulation and analysis are connected in transit across compute nodes using a client-server model. Data is transferred in a streaming manner, without disk I/O in between. The framework fulfills the presented use case requirements. Dedicated experiments on algorithmic solutions of the data transfer show, that no data is lost during transfer.

The design of the framework enables future integration of other software and thus could serve as a basis for in transit coupling in neuroscientific workflows on HPC systems.

Contents

List of Figures

1. Introduction

Today's high performance computing (HPC) systems are complex machines with hundreds of thousands of processors and deep memory architecture. This complexity enables end-users of these machines to formulate scientific goals such as weather forecasting, mathematical modeling or neuroscientific simulations [23, 20, 24].

However, in the era of big data, one of the main limiting factors is access to storage. With ever growing amounts of data produced by simulations, there is an increasing imbalance between processing power and I/O capabilities. Current computational capabilities enable extreme-scale scientific simulations and experiments that can generate much more data than can be stored at a single site.

One approach to address this issue on HPC systems is the online coupling of subsequent tasks within a workflow, while optimally using the systems computational resources. Online coupling uses techniques that perform on memory-resident data, instead of accessing storage or increasing or accelerating I/O performances [1, 33]. There has been a substantial amount of work in the last decade on the development, comparison and adaptation of online coupling techniques on HPC platforms, in particular with focus on *in situ* and *in transit* coupling [30, 34, 5, 32]. However, building a framework for an in situ and in transit coupling of subsequent tasks within a workflow poses several technical challenges.

The first challenge is to meet the requirements regarding speed, accuracy and memory demands. Especially for extreme-scale simulation and subsequent analysis, an important aspect is to handle the data flow and communication between tasks.

Workflows often consist of numerous computational tasks which have to be managed. Next generation HPC systems have focused on modular supercomputing architectures to handle such workflows [8, 9]. The second challenge includes the definition and scheduling for correct execution of those tasks as well as the configuration of the information exchange between them. A coupling framework could absorb parts of this complexity and thus not only address the required levels of usability, but also the modularity of such workflows.

A third challenge for such a framework is to provide an interface for interactive supercomputing and visualization of live results. The possibility of exploratory data analysis and interactive visualization on supercomputers is of great interest for end-users. However, in the context of HPC, analysis and visualization may use

different computing resources and have different requirements for end-user interaction. Therefore, such extensions need to be carefully prepared and managed.

In this thesis, an in transit coupling framework is developed and implemented to address these challenges for a use case consisting of a neuroscientific simulation and analysis. Chapter 2 outlines this use case and provides background information. The concept of in transit coupling is introduced and previous work on coupling frameworks is listed and distinguished from the work in this thesis. Scientific use cases of a specific analysis toolkit and their requirements are compiled in chapter 3. They have been chosen and formulated in collaboration with scientists and developers of this software. For the design of the framework, four key concepts, specific to the use case of coupling neuroscientific simulation and analysis, have been identified. Chapter 4 explains these key concepts in detail. They are used to describe specific challenges of the data transfer from a neural simulator to its analysis, namely data reception, data translation, data conversion and sending. These concepts and the scientific use cases from the previous chapter lead to the requirements for the framework. Finally, the framework is designed, implemented and tested on an HPC system. A description of the implementation and the technical details can be found in chapter 5. Furthermore, an experiment on two implementations of the data translation was designed and performed, the results of the implementation and the experiment are presented and evaluated in chapter 6.

2. Background and Related Work

This chapter provides background information on related software and technologies, as well as previous work. Furthermore, the terminology and concepts used in this thesis will be established. Section 2.1 and section 2.2 contain more details on neuroscientific simulation and analysis. In section 2.3 the in situ and in transit paradigms are introduced and in section 2.4 the coupling of workflow tasks in previous work is discussed and compared to the use case of this thesis.

2.1. Neuroscientific simulation

Computational neuroscience is a branch of modern neuroscience which involves theoretical analysis and abstractions of the nervous system. An important part in computational neuroscience is the simulation of the human brain. It allows to investigate mathematical models and theories and validate data from experiments. However, the human brain is a complex organ. It contains 86 billion neurons each with an average of 7,000 connections to other neurons [19]. Current compute power is insufficient to simulate an entire human brain on a molecular level. Thus, the simulations have to be done on multiple scales, ranging from molecular through the sub cellular to cellular level and up to the whole organ. Common spiking neural simulators are NEST [14, 25], ARBOR [2], NEURON [18], GENESIS [7], and BRIAN [15]. A comparative study of these is done by Tikidji-Hamburyan et al. [31]. Other simulations use macroscopic virtual brain models for seizure prediction and surgery [29].

2.1.1. NEST - Neural Simulation Tool

In this thesis, NEST is used as exemplary neural simulator on a cellular level. NEST is and has been an integral part of many projects, such as the SMHB [16] and HBP [19]. For the past 20 years, the development of NEST has been driven by scientific questions and has been following current computer architectures. Prominent examples include studies on spike-timing dependent plasticity in large simulations of cortical networks, the verification of mean-field models, models of Alzheimer's and Parkinson's disease and tinnitus[1]. Among many other features, the scaling ability of

[1] taken from `https://www.nest-simulator.org/`

NEST on supercomputers made it an obvious choice as neural simulator for the use case of this thesis.

For an in transit coupling framework, the output and input format of the coupled tasks is an important information. In this case, the output format of the simulator is of interest. In NEST, the spiking activity of the simulated neurons is encoded in events. Each event contains the information of a *neuron id* and a *spiketime*. Dependent on the parameters of the simulation (number of neurons, firing rate, etc.), the output rate of these events differ. For later experiments and tests (described in section 5.6), users of NEST were interviewed for estimations of typical rates.

2.2. Analysis of neuroscientific simulations

For analysis on simulation data in general, the visualization of its results is often included. In this thesis, visualization is considered separately, as additional task in a workflow. Furthermore, for the use case of coupling analysis with neuroscientific simulation, the specific input formats and type of analyses are of interest. For example, statistical analysis on spiking activity or membrane voltages of neurons in form of time series.

2.2.1. ElePhAnT - Electrophysiology Analysis Toolkit

Elephant [11] serves as the exemplary analysis toolkit for the neuroscientific use case in this thesis. It is actively developed by researchers at the Institute of Neuroscience and Medicine (INM-6) at the Forschungszentrum Jülich. Elephant provides generic analysis functions for spike train data and time series recordings from electrodes.

Close collaboration with users and developers of Elephant made it possible to formulate scientific use cases and requirements for the in transit framework. They are described in more detail in chapter 3. In the same way as for NEST (section 2.1.1), the input format of Elephant needs to be defined for a successful data transfer between these tasks. Here, the above mentioned spike train data are the equivalent to a list, containing the spiking activity of a neuron.

2.3. In situ and in transit terminology

In the traditional usage of simulation, analysis and visualization, data has been written to disk and read back into memory for further processing. This is sometimes referred to as *post hoc* processing. The most prominent terms for techniques of avoiding I/O, i.e. the online coupling of tasks within a workflow, are *in situ* and *in*

transit. However, many applications and use cases in science for in situ and in transit have led to a diverse terminology. The following synonyms are commonly found in literature:

- *in situ processing* ↔ *in transit processing*
- *co-processing* ↔ *concurrent-processing*
- *tight coupling* ↔ *loose coupling*
- *on-node processing* ↔ *off-node processing*

Even though this terminology seems to be clear, it is still used inconsistently.

In this thesis, the two umbrella terms of in situ and in transit are used. Both terms describe the contrast to post hoc data processing. Furthermore, they distinguish *where* the data is processed when coupling two workflow tasks, i.e. simulation and analysis.

In situ coupling

In the in situ coupling paradigm, simulation and analysis typically share the same compute node or even the same cores (see figure 2.1). In situ is also referred to as tight coupling [3, 10] or co-processing [26].

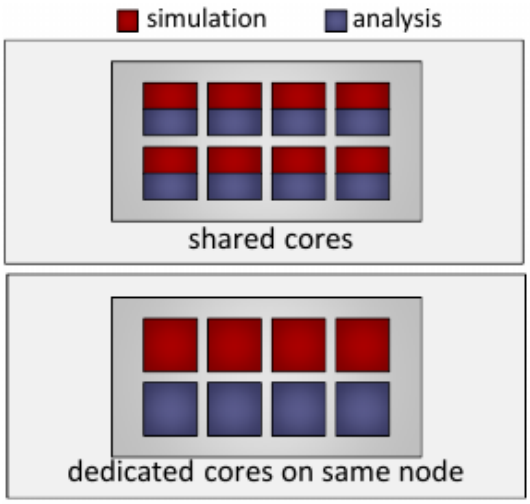

Figure 2.1.: In situ coupling. Simulation and analysis share the same cores or the same node on an HPC system. *Reprinted from [6].*

This coupling mechanism does not only avoid writing to disk, it also avoids data transfer across compute nodes or even across different machines. It enables frequent data exchange between the coupled tasks at the cost of flexibility (available memory, compute resources, etc.).

In transit coupling

In transit coupling or, in analogy to above, *loose coupling/concurrent processing*, distributes the coupled tasks to separate, dedicated nodes or external machines (see figure 2.2). In this case, data from simulation is transferred over the network to analysis.

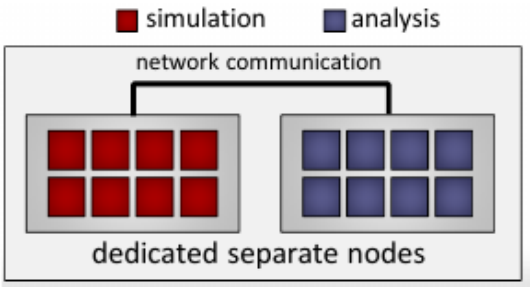

Figure 2.2.: In transit coupling: Simulation and analysis have dedicated nodes on the same machine or even external resources. *Reprinted from [6].*

The network communication obviously limits the amount of data that can be transferred. Therefore, in transit coupling favors less frequent data exchange between the coupled tasks. However, intermediate processing during data transfer can be utilized to reduce this bottleneck, as well as used for asynchronous processing. This indirect data access is an advantage of in transit over in situ coupling. Additionally, the dedicated resources can be highly specialized for the tasks.

2.4. Previous work on in situ and in transit coupling

This section provides comparison factors for in situ and in transit coupling and an overview of previous work. The aim is to describe different approaches and also distinguish them from the work in this thesis.

2.4.1. Comparison factors

The *In situ Terminology Project* [21] formulated six factors to characterize in situ and in transit coupling frameworks: Integration Type, Proximity, Access, Synchronization, Operation Controls and Output Type.

These factors are used for comparison of the in transit coupling framework in this thesis with related work done in the past.

Integration Type ranges from directly embedding analysis routines into simulation

code to indirect connection of simulation and analysis, for example with third party libraries.

Here: Simulation and analysis are connected indirectly, no code customization of NEST or Elephant is done. The indirect connection is also done without the dependency on a third party library.

Proximity is the main comparison factor between in situ and in transit. Even though this is not a binary measure, close proximity often refers to (on-node) in situ coupling, while far proximity refers to (off-node) in transit coupling.

Here: Far proximity. Simulation and analysis are executed on different nodes, but still on the same machine.

Access to the simulation data can be either direct access to the same memory or indirect access via a communication mechanism that copies the data to a separate memory.

Here: Indirect access. Simulation and analysis do not share the same memory. Far proximity is commonly connected with indirect access.

Synchronization describes how simulation and analysis can operate with respect to each other. In synchronous coupling, simulation and analysis share the same computing resources and execute only one at a time. Asynchronous coupling refers to concurrent processing.

Here: Asynchronous coupling, simulation and analysis are executed concurrently.

Operation Controls describe the interactivity of the coupling, ranging from fixed analysis operations to modifications by end users during simulation.

Here: A hybrid model of fixed analysis and end user interaction.

Output Type differentiates explorable and non-explorable output. It has no effect on the mechanism of the coupling, but may be an important descriptor for end users.

Here: - (output needs to be visualized).

2.4.2. Previous work on in transit coupling

Previous work on in situ and in transit coupling resulted in the development of numerous frameworks. A detailed and comprehensive study on in situ methods can

be found in Bauer et al [4]. However, the above evaluation of the comparison factors for the framework in this thesis excluded in situ solutions. Furthermore, a direct comparison of in situ and in transit has been done in Kress et al [26]. They argue that in transit will play an important role in HPC in the foreseeable future and also addresses many of the current issues of in situ.

Thus, some selected in transit solutions are presented here for comparison with the work in this thesis.

The first obvious in transit specific comparison factor is proximity. But as shown in Aktas et al [3], *far proximity* does not have to mean only off-node. In their work, a data transport service is designed for scheduling and controlling of the transport resources of a network. In their case, far proximity includes also other machines or even locations.

Marrinan et al [27] compares three techniques of data transfer between distributed memory applications. They make use of a parallel file system and network-accessed shared memory to temporarily store data. The difference to the work in this thesis is a direct integration of simulation and analysis and the use of indicator states in files for a synchronized coupling.

A hybrid approach to in situ and in transit was taken by Bennet et al [5]. They explore the design and implementation of three common analysis techniques, namely topological analysis, descriptive statistics and visualization. The analysis is then subdivided into two stages: first in situ data filtering and aggregation, second in transit analysis. They make use of the third party libraries DART and DataSpaces for task scheduling and resource management of the analysis. The aim of the framework in this thesis is to avoid data manipulation, i.e. filtering, and be independent of third party libraries.

In Jin et al [22] the framework of [5] is extended for 'dynamic change of data volumes and distributions'. This is specifically occurring in Adaptive Mesh Refinement (AMR) based simulations. They use a cross-layer approach for different spatial and temporal resolution adaptations at runtime. In this thesis, their approach is not needed, due to a fixed temporal resolution.

A client-server approach has been done by Usher et al [32]. Their solution has many similarities to the design of the framework in this thesis. However, one important difference needs to be mentioned. The simulation poses as server and processes queries from the client, i.e. analysis. This is done non-blocking but nevertheless produces overhead. The approach in this thesis is, to have the simulation output completely independent from analysis requests.

3. Use Cases

The design and development of the in transit coupling framework described in this thesis is motivated and driven by scientific use cases from end-users of HPC systems. The aim of the use cases presented here is to give an example for the range of requirements in transit coupling of a neuroscientific simulation and analysis has to expect and fulfill.

The analysis toolkit Elephant (section 2.2.1) provides a rich set of analysis methods to choose from for the definition of use cases. In collaboration with users and developers of Elephant, the following two use cases were formulated: A) Statistical measures of spike train data on streaming output of a simulation. B) Interactive visualization of correlations in a simulated neural network.

3.1. Statistical measures of spike train data

This use case has been chosen for examples of fast and continuous analysis. The `statistics` module of Elephant provides statistical measures of spike trains and functions to estimate firing rates. An online analysis and real-time visualization of statistical measures on streaming output of a simulation would be a useful tool for scientists. It would for example allow an immediate identification of interesting subsets of the data for more complex analyses or parameter changes of the simulation.

Figure 3.1 shows a mockup of such a live-visualization of statistical measures for streaming data. An incoming stream of spiketimes of neurons (spike trains) from a simulation of a neural network is visualized. Statistical measures, such as mean firing rate or coefficient of variations can be calculated and displayed. In the context of streaming data from a simulation to analysis, this use case benefits from fast data transfer. In order to avoid data loss, the rate of incoming data at analysis-side should at least match the rate of simulation output. Missing data because of a slow transfer between simulation and analysis would possibly lead to skewed, if not incorrect results.

3.1.1. Requirements

The following list of user requirements have been identified for this use case.

1. The user should be able to specify the number of neurons to analyze.

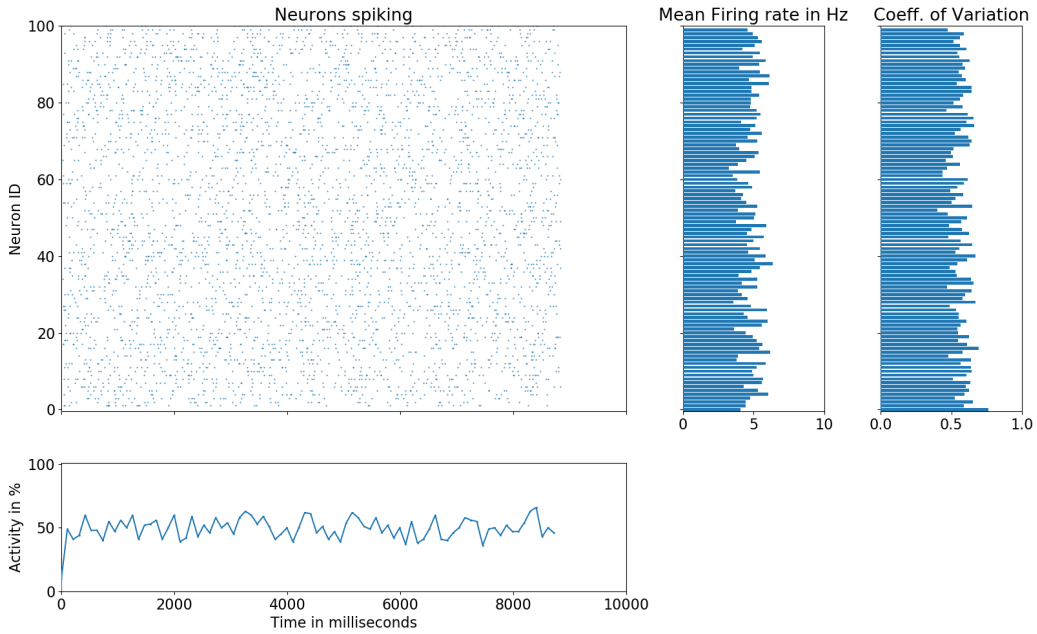

Figure 3.1.: Mockup: Live-visualization of streaming data from a simulation in form of spiking activity of neurons and examples from the statistics module of Elephant in form of mean firing rate and coefficient of variation.

2. The user should be able to specify the time frame to analyze.

3. The user should be able to conduct various statistical analyses on the data.

4. A fixed set of analyses should be provided, as shown exemplarily in the mockup design above. This would allow users to spot interesting subsets of the data for further analysis

5. The data from simulation should be transferred to analysis as fast as possible. This is important to enable the Elephant module to compute and present the results timely.

6. No data should get lost during transfer.

This presented use case and most of its requirements are also valid for future work on interactivity of the framework.

3.2. Incapy - Interactive Neural Correlation Analyzer for Python

The *Incapy project*[1] is developed at the Forschungszentrum Jülich. It can visualize the correlations between neurons from spike train recordings. The interactive visualization facilitates the analysis of the data and enables exploratory data analysis.

The `incapy` module uses correlation coefficients of pairwise correlated neurons and position parameters as input (figure 3.2 A). A force-directed graph drawing algorithm is used to continuously update the position of the neurons (figure 3.2 B), until an end state is reached (figure 3.2 C).

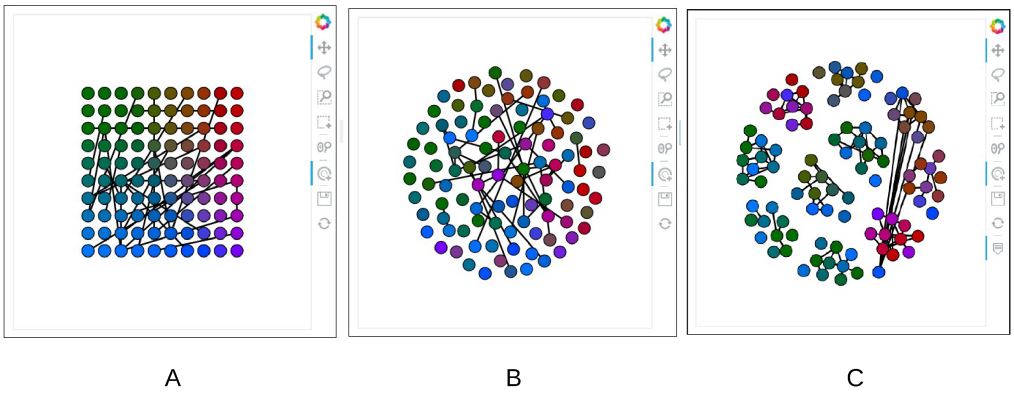

Figure 3.2.: The visualization of the Incapy-Software. A snapshot of the UI with the correlation graph at the start (A), during runtime (B) and at a reached end state (C) is shown.

Live visualization and exploratory data analysis of streaming data in an in transit workflow environment is an interesting use case. However, the integration of Incapy into such an environment poses the challenge of temporary data storage in memory. Incapy was implemented to have recorded data from file as input. Instead of a continuous data transfer as in the use case above, the spiking activity of n neurons over m seconds is required at once for the calculation of a cross-correlation matrix. This challenge was discussed during a collaboration with the developers of Incapy and a set of requirements for the integration into an in transit workflow was compiled.

[1] https://github.com/INM-6/incapy

3.2.1. Requirements

Some of the compiled requirements contained extensions and customizations of the Incapy project. Listed below are only the requirements, the in transit coupling framework has to fulfill.

1. The user should be able to specify the number of neurons to analyze.
2. The user should be able to specify the time frame to analyze.
3. Additional parameters (required and optional) should be set beforehand. For example, position and labels of the neurons or correlation threshold parameters.
4. Temporary storage and transfer of chunks of data is needed for the calculation of cross-correlation matrices with Elephant.
5. The visualization with Incapy should still be interactive.

4. Design

The combination of extreme-scale simulations and in transit coupling poses the main challenge in the design of the framework. Extreme-scale simulations imply a large amount of data in a short time. An in transit coupling framework, which handles the data flow between tasks of a workflow, needs to transfer this data and at the same time fulfill the requirements to speed and accuracy. Additionally, the modularity and developer usability of the framework should not be neglected. In this chapter, key concepts of the framework are introduced and possible solutions are discussed. The design of the framework is derived, with regard to those key concepts and the use case requirements from chapter 3.

4.1. Key Concepts

There are many important factors relevant to end users, computer scientists and developers of simulations and analyses in HPC. Below, some of the factors more relevant to the design and development of the framework in this thesis are listed:

Data Access The available data that can be accessed by analyses in an in transit environment, depends on the amount of data that can be temporarily stored in memory and transferred to analysis. There are multiple variations of data access required by analyses, ranging from low to high frequency and from shorter to longer time frames. Thus, the limitations of temporary data storage and transfer restrict data access.

Data Movement Moving large amounts of data between workflow tasks or compute nodes of the same program on a supercomputer, is expensive and should therefore be kept to a minimum. Intermediate computations, data conversion and data reduction should be considered to only move the necessary amount.

Data Translation There are many different data formats and structures of simulation output and analysis input. The translation of these formats in an in transit environment has two general options. The first option is to enable the analysis to work directly on the simulation output. This would include writing custom code for analysis and simulation. The second option is to keep the respective data

structures and layouts and copy the data from the simulation data structure into the analysis data structure.

Resources In transit coupling of two workflow tasks requires additional resources for intermediate computations and data exchange. But resources for supercomputing are in high demand and expensive. Therefore, these additional resources add to the costs of running a simulation and an analysis. However, the data can be transferred to specialized parts of a supercomputer, that can be used efficiently and asynchronously.

Scalability Dependent on the amount of computation and communication needed, an in transit coupling framework does not scale to the levels of simulations (e.g. hundreds of thousands of cores). This restriction should be considered in contrast to the advantage of independently scalable simulation and analysis.

Developer Usability Developing an easy to use in transit coupling framework with use case flexibility is challenging and contains a wide range of topics to consider. To allow for topics such as development, deployment and dependencies to be more easily integrated in the future, changes in simulation or analysis code or dependencies on third party libraries need to be carefully managed or avoided.

From these factors, four key concepts can be derived. They are essential for the use case of in transit coupling of a neuroscientific simulation and analysis and will be described in more detail in the following four sections.

4.1.1. Receive and store data

Data produced by simulation is expected as input to the framework in a streaming manner. Thus, the framework has to receive and forward this data to analysis for further processing. The main challenge is to handle the amount of incoming data with the above mentioned data access and scalability in mind. This means to receive data fast enough and temporarily store it for as long as possible. Since this data reception and temporary storage are possible bottlenecks, a solution should be extensible for parallel data reception and forwarding.

4.1.2. Asynchronous data transfer

An important concept of an in transit coupling framework is asynchronous data transfer. Simulations running at different scales (frequency, size) result in different rates and amount of data to be transferred to analysis. The rate of data transfer

to analysis is on the one hand limited in speed by the software (implementation, algorithms, etc.), and on the other hand dependent on end-user demands. A synchronized data transfer between simulation output and analysis input, would result in either data loss (slower than needed) or unnecessary data movement (faster than needed). Therefore, data transfer should be asynchronous, i.e. flexible for different rates of incoming and outgoing data.

One obvious minimal transfer rate desirable to achieve is to receive, store and send out data before it is overwritten by new arriving data due to lack of memory. In other words, to not lose any data during the transfer. The use case described in section 3.1 for example has this transfer rate as requirement. In case of a transfer rate faster than the incoming rate, the data transferred to analysis would contain duplicates. In this first design of the framework, the upper limit of data transfer is not set, i.e. duplicate data is allowed. This allows to measure the performance of the implementation in later experiments (see section 5.6).

4.1.3. Transpose data

Data translation between neuroscientific simulation and analysis can be done in both ways described above, either customize simulation and analysis code or copy the data between data structures. However, the first option of customizing the simulation or analysis data layout has been discarded. It would allow the direct connection in this specific case, but requires code customization of an established simulator or analysis toolkit. Furthermore, it would impede the use of other simulations or analyses in the future and, as briefly described in Kress et al. [26], there are different approaches to similar issues used by the community so far.

For the use case of neuroscientific simulation, the second option of data translation was chosen. Even though it involves to copy the data, which obviously is undesirable, there are two arguments in favor of this option. First, it avoids code customization on simulation and analysis side. It enables the use of schemes, interfaces, data models and conventions in the future. Second, the output of many neuroscientific simulations consists of a series of events. Each event is a pair of dates, e.g. *(id, time)*. Additionally, analysis on such data often expects input in the shape of time series. This means, re-usability in this specific case of neuroscientific simulation and analysis.

In the case of NEST, each event of the output consists of a *neuron id* and a *spiketime* (see section 2.1.1). Analyses, in this case Elephant, on such data expect input as a time series of spikes per neuron, so called spiketrains. Therefore, these

events have to be 'transposed'. Transposing in this context means that all events, in a given time frame and for a given number of neurons, are sorted by neuron id and have the respective spiketime collected in a list. After transposing, each neuron with its unique *neuron id* has its own *list of spiketimes* (see figure 4.1)

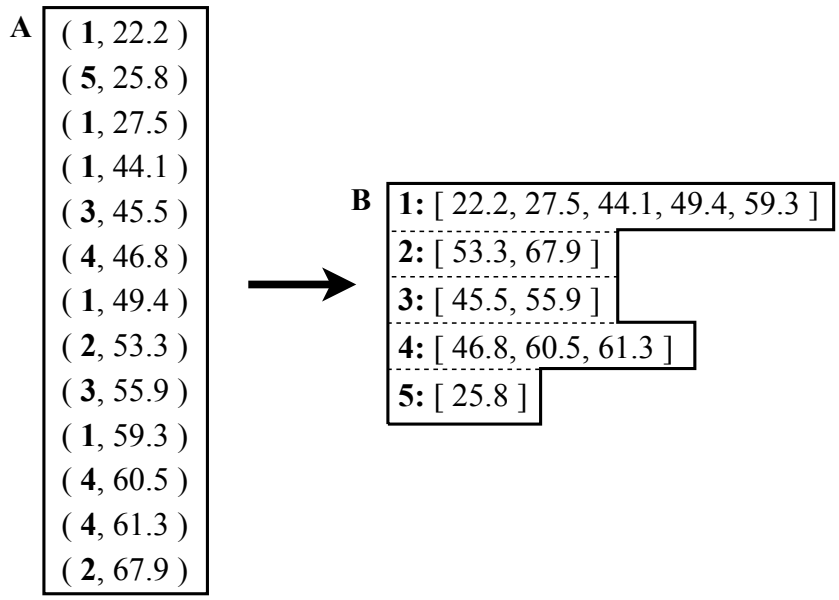

Figure 4.1.: **A** The table represents an output of a NEST simulation, each row is an event with *neuron id* and *spiketime*. **B** The transposed data, each row contains all spiketimes of one neuron.

This task poses two major challenges for the implementation, when considering the data from simulation as incoming data stream:

An unknown number of events per neuron. The size of the simulation (number of neurons) and the amount of data to transpose (number of total events) both influence the number of events per neuron. This makes it difficult to find a suitable data structure, since the transposed data does not have to be of rectangular shape, i.e. in matrix form (see figure 4.1). Additionally, memory allocation for a data structure of unknown final size, as well as frequent memory access by adding new events should be avoided as much as possible.

The time needed to transpose. A fast transposition of the data is critical to ensure a data transfer from simulation to analysis without losing data (see section 4.1.2). Also, the transposes have to be done both frequent and with varying amounts of data, dependent on analysis requests and available memory.

4.1.4. Exchange, convert and send data

The transposed data has to be further processed and, if done in parallel, exchanged between processes before sent to analysis. Conversion into a suitable and efficient datatype is necessary for optimal usage at analysis site. Even though it may be specific to different analyses, the conversion to a datatype before sending is easily extensible. Additionally, it fits the chosen option of data translation (see section 4.1.3), which keeps the respective data types of simulation and analysis.

Figure 4.2 summarizes this parallel workflow. In case of parallel transposing, the

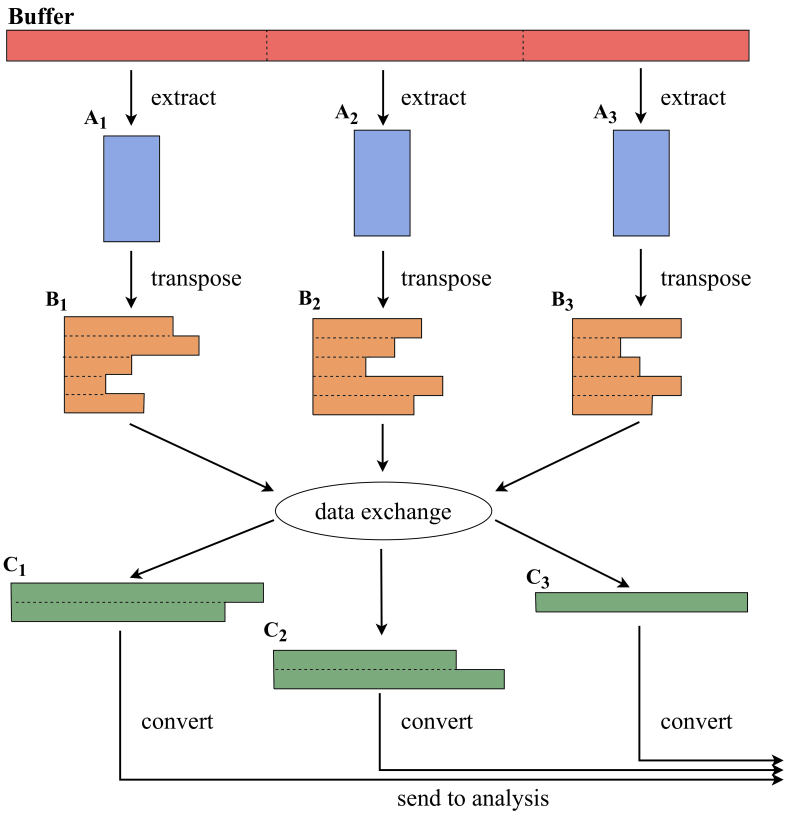

Figure 4.2.: The parallelized workflow. $A_1 - A_3$: Data from buffer is extracted and transposed in parallel. $B_1 - B_3$: The spiketimes of each neuron are distributed among the processes and need to be exchanged. $A_1 - A_3$ and $B_1 - B_3$ correspond to A and B in figure 4.1. $C_1 - C_3$: After the data exchange, all spiketimes of the respective neurons are collected. Finally, the data can be sent to analysis.

data exchange between processes is a critical step during the data transfer. As explained in section 4.1.3, the output of neuroscientific simulations arrives over time

and is thus split and distributed among the processes in the time domain (figure 4.2: $A_1 - A_3$). Each process then performs the transposition in parallel (figure 4.2: $B_1 - B_3$). $A_1 - A_3$ and $B_1 - B_3$ correspond to A and B in figure 4.1. Consequently, the spiketimes for each neuron are distributed across processes and have to be assembled before converting and sending them to analysis (figure 4.2: $C_1 - C_3$). This means the data has to be either collected on one process or redistributed among the processes .

For extreme-scale simulations, memory and runtime issues are expected when collecting and concatenating all transposed data on a single process. In order to avoid this, the data exchange needs to be implemented.

Finally, the converted data can be sent to the analysis.

4.2. Requirements and design of the framework

This framework will be a component in a larger workflow environment in the future. Thus, the design also contains related work previously done in the Simulationlab Neuroscience [12] on the simulation side and partly future work on the analysis side. Both parts may be subject to change.

The main focus of this section lies in providing a basic design for in transit coupling for the specific use case of transferring data from NEST as neural simulator to Elephant as analysis toolkit. This design will then be implemented and tested in an HPC environment.

A first set of user requirements is compiled, based on the use case requirements listed in section 3.1.1 and section 3.2.1:

U1: Fast data transfer The data from simulation should be transferred as fast as possible.

U2: Flexible data transfer The amount of data sent to analysis should be flexible.

U3: Flexible data access The user should be able to request data flexible, at different rates and times.

U4: Exchange of parameters The framework should enable an exchange of parameters and settings before the simulation (the data transfer) starts.

U5: State of the simulation Information about the start and the end of the simulation should be exchanged between simulation and analysis.

U1 is derived from the requirements of section 3.1.1. Achieving the minimum transfer rate is necessary to avoid data loss. Additionally, more complex analysis needs compute time itself, so fast data transfer is desirable. U2-U4 are connected over several requirements from chapter 3. The user wants to set parameters beforehand (U4) and specify what data to analyze (U2). Flexible data access (U3) is for example needed in use case of the Incapy integration, when the user wants to change the size of the cross-correlation matrix to visualize or change the frequency of the request. U5 is important for parameter exchange beforehand and the information that no new data will arrive.

A second set of technical requirements is directly derived from the key concepts (section 4.1.1-section 4.1.4) discussed above:

T1: Databuffer The received data should be stored in a buffer. This is necessary to allow intermediate processing and transfer in chunks of data rather than a 1:1 mapping from incoming to outgoing package sizes.

T2: Arbitrary data access As described in section 4.1.2, the framework should enable data access at arbitrary rates, i.e. asynchronous data transfer.

T3: Transposition of data The data has to be transposed as shown in figure 4.1. This is a critical step and possible bottleneck. Therefore, a parallel solution should be implemented and tested.

T4: Exchange data A critical step after parallel transposition. The data has to be redistributed according to figure 4.2. This is also a possible bottleneck because of the amount of communication needed.

T5: Convert data The transposed data should be converted to a suitable datatype for analysis. This should be done in a modular way to be extensible for other data types and analyses.

T6: Send to analysis This completes the coupling of simulation and analysis. Following T4 and T5, this also can be done in parallel.

Requirements T1 (receive), T3 (transpose) and T6 (send) are critical to provide in transit coupling of a neuroscientific simulation and analysis. Requirements T2 and T5 are important for modularity on the analysis side. T2 also corresponds to flexible access rates and requests, which fits requirement U3. Requirement T4 is a consequence of parallel data transposition and its implication will be further discussed in section 5.6.

To summarize, figure 4.3 shows the design with the technical requirements highlighted. The in transit coupling mechanism is defined as *in transit server*, simulation

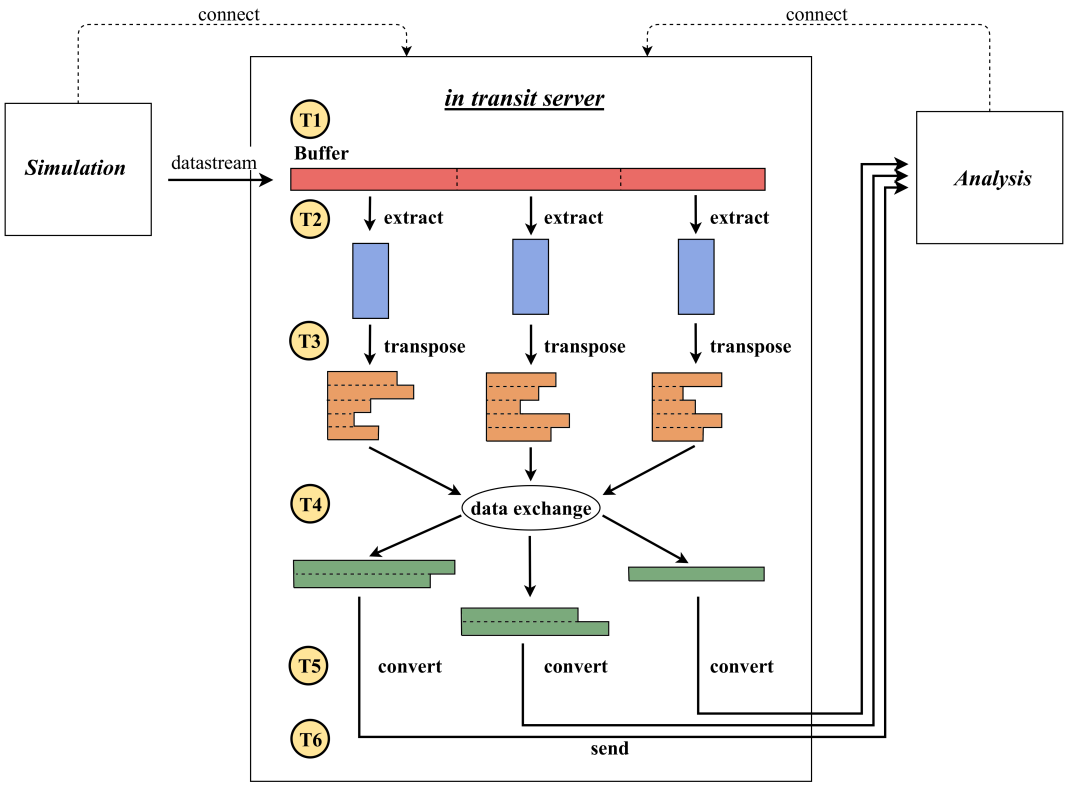

Figure 4.3.: The design of the framework. The technical requirements T1-T6 are highlighted. Simulation and Analysis clients connect to the in transit server. A data stream from simulation is received into a buffer (T1). Asynchronous access to the buffer (T2) can be done in parallel. Data transposition (T3), exchange (T4), conversion (T5) and finally sending (T6) to analysis is handled by the server.

and analysis are defined as *clients*. In this setup the server enables data transfer and at the same time does not rely on simulation and analysis to share the same resources, i.e. to be tightly coupled. The server provides simulation-side and analysis-side interfaces for connection and parameter exchange as well as the overall data transfer mechanism. The features shown in figure 4.3 can be implemented on server-side in an extensible way. No changes on simulation- or analysis-side are required. This approach requires additional effort in the future to support for example different I/O behaviors or new data types, but simulations and analyses do not need to be customized. The implementation and solution to the technical requirements T1-T6 can be found in chapter 5.

5. Implementation and Experiment

The implementation of the framework follows the design from section 4.2. This chapter will describe the technical details, how the requirements are met and how the framework handles the data flow from simulation to analysis. Finally, an experiment on two different implementations of data transposition algorithms will be described.

With Elephant being written in Python and NEST providing a Python based user interface, Python (v.3.6.5 locally, v.3.6.8 on JURECA) was used as programming language.

5.1. Client-Server model and communication with MPI

The Client-Server model was implemented using MPI 3.0[1], specifically mpi4py in version 3.0.1 on JURECA. At first, the in transit server uses MPI to open a TCP/IP port and to establish a network address at which it will accept connections. The respective MPI functions are `MPI_Open_port` and `MPI_Comm_accept`. This information is then published to the simulation and analysis clients via a file. Once the connections are made, a new MPI intercommunicator between server and client is created (see figure 5.1).

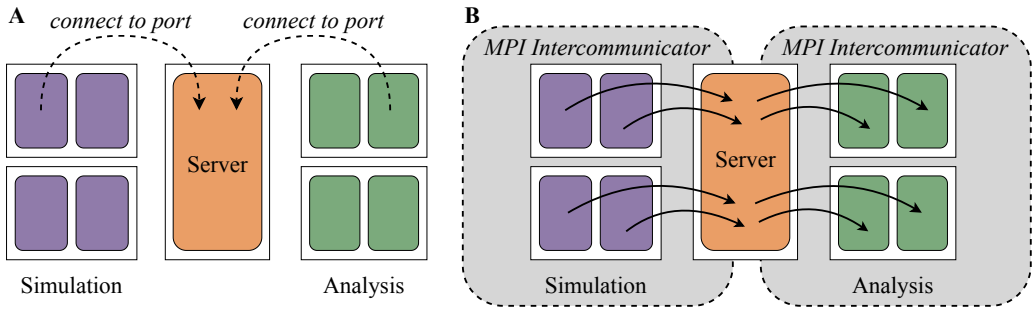

Figure 5.1.: MPI Client-Server model. **A** The server opens TCP/IP ports, where clients, i.e. simulation and analysis can connect to. **B** After the connection is made, communication via an MPI intercommunicator is possible.

However, there is an important detail to be noted if the server is running with multiple MPI processes. The port information which is used to make the connection is encoded in a system specific string. This string contains information about the

[1] https://www.mpi-forum.org/mpi-30/

MPI process which created it. The `MPI_Comm_accept` call uses the port information to allow a client connection, but also has to be done collectively. This means, that all processes have to have the same port information when making the `MPI_Comm_accept` call.

5.2. Python modules

The complete proof of concept is organized in the following Python modules:

server.py starts the in transit workflow. It accepts connections from analysis and simulation and provides the interface to exchange information and parameters, e.g. number of neurons. The server makes calls to the methods of the `data_handler.py` module for data reception from simulation.

simulation.py is responsible for producing the data stream coming from a NEST simulation. This is achieved by simulating a NEST simulation. The module simply reads the output of a NEST simulation, which has been run in advance, from file. The number of lines in the file, i.e. spike events of the neurons, depends on the parameters of the simulation. After the connection to the server is established, parameters can be exchanged. For example information about the size and length of the simulation.

analysis.py is the receiving end of the in transit coupling and serves as the interface for analysis requests. It also establishes the connection to the server and exchanges important parameters and information. More detailed examples of such information are described in section 3.1 and section 3.2. The transposed and converted data from the server is received in a stream and analysis can be directly performed on the data. The module is kept extensible for future work on interactive analysis and visualization.

data_handler.py handles the logic during the in transit data transfer. It provides the functionality described in the design (section 4.2). There are four parts internally: 1) data reception 2) data transposition 3) data exchange 4) data conversion and sending. All of them operate independently and can therefore be extended and/or exchanged in the future. A more detailed description of these parts can be found in the following sections.

5.3. Databuffer and asynchronous data transfer

The data reception and temporary storage was implemented using a ring buffer. This buffertype provides a fitting *first-in-first-out (FIFO)* logic, to handle the incoming data stream. Moreover, it is easily handled by knowing the size and keeping track of the current head of the buffer.

To solve the task of asynchronous data transfer (see section 4.1.2), data reception and further processing had to be separated. The receiving process(es) and the process(es) which continue working on the data, must not be dependent on each other. In an MPI program, as in the present case, this usually means additional communication. Since MPI 3.0 there is another option, which is used in this thesis:

MPI: One-sided communication This type of communication extends the communication mechanism of MPI. Point-to-point and collective operations require the co-operation of a sender and receiver. One-sided communication allows processes to read or write data without the other processes' involvement. In MPI terms this means a process declares a *window* in its memory, which is accessible by other processes. In `mpi4py`, this is done by calling the function `MPI.Win.Allocate_shared(size,datasize,comm)`. This call is collective and therefore executed by all processes of the communicator `comm`. On each process, memory of `size` bytes will be allocated (`size` can be zero). The function returns a handle to the window (`win`), i.e. the locally allocated memory. The handle can then be queried by the other processes by calling the function `win.Shared_query(root)`. A pointer to the window of process `root` is returned, which can be used as shared memory buffer. In a final step a data structure (e.g. a `numpy array`) is created, whose data points to the buffer.

The possibility to separate writing and reading processes can now be used for asynchronous data transfer. The receiving process(es) can allocate the shared memory buffer to write data into and give reading access to the process(es) involved in the data transfer. Since this is a one-sided communication, reading and writing process(es) do not depend on each other.

In summary, data reception, storage and asynchronous transfer was implemented by the combination of MPI shared memory and a ring buffer. Additionally, the memory access logic of a ring buffer can easily be optimized or alternative buffertypes can be implemented.

The current implementation dedicates one process for data reception and writing into the buffer. Experiments with parallel receiving and writing into buffer, as well as

multiple buffers will be subject of future work. Buffer access and further processing can be done with multiple processes as described in the following sections.

5.4. Data transposition

The implementation of the data transposition had to address the following important issues:

1. Data extraction from the buffer needed to be done in parallel.
2. The transposition problem itself as described in detail in section 4.1.3 needed an efficient solution.
3. Exchange of data after transposition, i.e. communication between processes (see next section).
4. The end-of-simulation signal had to be handled and forwarded to analysis.

The first issue, buffer access and data extraction could be implemented embarrassingly parallel. The pointer to the shared memory can be passed as buffer argument when the above mentioned `numpy array` is initialized. Therefore, accessing the array directly exposes the raw data in the shared memory[2]. Conveniently, this access is made by calling the `numpy.array_split()` method, which evenly splits the data among the processes for parallel transposition.

The following two parallel solutions were implemented. They were tested and compared for their runtime and data throughput. This experiment will be described in more detail in section 5.6

Solution 1: Append This transposition algorithm makes use of Python lists and the `append()` method. It is the most straightforward way of solving the transpose problem, since it exactly follows the second option of data translation in section 4.1: the data from simulation is directly copied into a data structure which can be used for analysis.

At first, `n` empty lists for `n` neurons from simulation are allocated and put into a list. This *list of lists* is a suitable data structure for non rectangular matrices. Each process then iterates over its data, i.e. events. If `(i,j)` is the event of neuron `i` spiking at time `j`, then `j` is appended to the i-th list. The result is the data structure shown in figure 4.1-B. However, this algorithm is heavy on

[2]https://docs.python.org/3.6/c-api/buffer.html

memory access and allocation, since each addition is potentially a new memory allocation.

Solution 2: Sort and split This second algorithm groups the events by neuron id in three steps. First, all events are sorted along the first axis (neuron id). Second, the number of spikes per neuron are counted and the events are split along the second axis (spiketime) accordingly. Third, empty arrays are added for neurons without any spikes.

In comparison to the first solution, this is a more complex algorithm. However, in order to have an acceptable runtime, several efficient numpy implementations can be used. One example being the `numpy.sort()` method, which runs in $O(n\ log(n))$[3]. Neuron ids and spiketimes are counted and grouped by calling `numpy.unique()` and `numpy.split()`. Another important difference is the amount of memory accesses and allocations. While the sorting, grouping and splitting of the data requires intermediate copies, no dynamic allocation by appending new data is necessary.

The last issue the data transposition had to address, was to receive the end-of-simulation signal and forward it to analysis. This was implemented with the use of MPI tags. After the simulation ended, the receiving process on server side is informed by using a different MPI tag. The server then uses its intracommunicator to broadcast that no new data will arrive in the shared memory buffer. Finally, the analysis client is informed via the same mechanism.

5.5. Data exchange, conversion and sending

After the parallel transposition additional communication is required before the data can be sent to analysis (described in section 4.1.4). This data exchange is an integral part of the in transit workflow of this thesis and it will be part of the experimental setup in section 5.6. Moreover, the data exchange is a complex step on its own and was encapsulated and implemented separately to enable optimizations and extensions.

The following implementations were tested to solve this communication and exchange problem:

Solution A contains one single `MPI_gather` operation to collect the distributed transposed data on one process.

[3] https://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html

Solution B contains n `MPI_gather` operations, with n being the number of neurons. All spiketimes of each neuron are collected one by one and then distributed equally among the processes.
Solution C contains p `MPI_gather` operations, with p being the number of available processes. The distribution of the neurons to the processes is done beforehand, thus avoiding several gather operations in comparison to solution B. This solution leads to the same result as solution B with less communication but adds algorithmic complexity and computations.

Both, A and B were discarded after initial tests. Solution A creates a bottleneck at the single process which gathers all data taken from the buffer or requested by analysis. Additionally, having all processes waiting for one process to complete its calculations is an inefficient use of the computing resources. Solution B produces massive communication overhead, while having the same result as solution C: Distribute the neurons among the processes and collect all spiketimes for each neuron on the processes. The current implementation follows solution C. Nevertheless, this problem needs to be further investigated in the future for optimal solutions.

Figure 5.2 shows the summary of the implemented in transit workflow in form of an example for three processes, including a detailed depiction of Solution C (one MPI gather operation for each process). The figure shows the additional computations which have to be done for this kind of gather operation, the transformation to one-dimensional shape with known size before the gather operation and the reshaping afterwards.

The arrays collected at each process after the data exchange contain all spiketimes of each neuron and are sorted by neuron id. In a final step, the spiketimes are converted into `numpy arrays` and sent out in parallel to the analysis client. On analysis side the arriving spiketime arrays (spike trains) can now easily be assigned to the corresponding neuron id, by knowing the id of the sending process and the total number of neurons.

5.6. Experimental setup

An important measure for a successful in transit coupling is the data throughput. That is, the speed and amount of data received and sent by the server.

In summary, the goal of the experiments was to test and compare the two parallel transposition algorithms implemented against their fulfillment of the requirements from section 4.2. The measure chosen was the amount of data the algorithms could transpose, convert and exchange, for a different number of processes. An additional

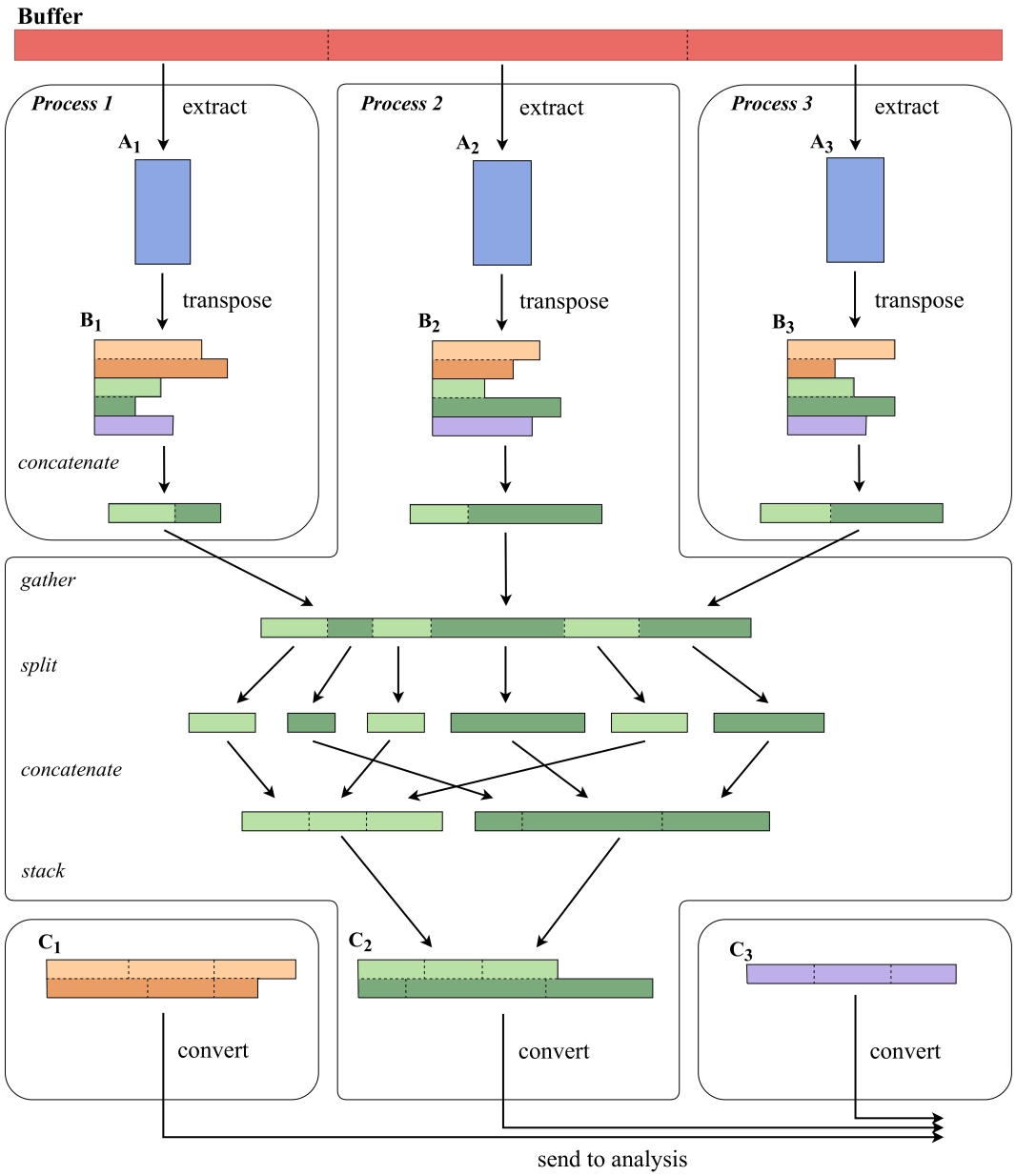

Figure 5.2.: Summary of the in transit coupling workflow, exemplary for three processes and five neurons. Each process extracts ($A1 - A3$) and transposes ($B1 - B3$) its data. The transposed data is colored according to the distribution of the neurons after the data exchange: two neurons for process 1 in orange, two neurons for process 2 in green and one neuron for process 3 in purple. The data exchange follows solution C: One MPI gather operation is performed for each process, here shown for process 2. The data of multiple neurons is gathered in a single one-dimensional array which, subsequently, has to be ordered by neuron id. Process 1 and 3 would perform the exchange respectively. After the data exchange, each process has all the spiketimes of its assigned neurons ($C1 - C3$).

measurement taken was the overall throughput of the workflow in relation to the size of the buffer. Also, by measuring the ratio of data received and sent, the key concept in section 4.1.2 and requirement T1/U1 (fast data transfer) are addressed. Below is a detailed description of the setup and the settings of the experiment.

In the context of the use case in this thesis, coupling a neural simulator and its analysis, a critical step during data transfer is the transposition of data. It has implications on other steps (as described in section 4.1.4) and is therefore expected to heavily influence the overall throughput. The first experiment consists of testing the two different parallel implementations of the data transposition, as described in section 5.4. They have algorithmic differences, which manifest in computational complexity and memory allocation. Furthermore, they have some minor differences in the amount of communication needed afterwards.

As mentioned before, this communication or data exchange (requirement T5) is a consequence of parallel transposition. If more processes take part in the parallel transposition, more communication to exchange the data has to be done. Figure 5.2 shows this relation between parallelism (number of processes) and amount of communication. The relation is expected to have impact on the results of the experiment. Thus, the data exchange after transposition was taken into account as part of the transposition during measurements.

In the second experiment, different buffer sizes were tested. They directly influence the performance of the algorithms, since the entire content of the buffer is distributed among the processes each time it is accessed. Thus, the result can be used to evaluate the performance of the algorithms with varying amount of data.

The setup on the JURECA supercomputer:

- All tests were run on the JURECA supercomputer (see appendix A for the configuration).
- In order to easily run the MPI server and the clients concurrently, interactive sessions[4] were used instead of submitting single jobs.

The setting used for the experiment:

- Each testrun started with both simulation and analysis connecting to the in transit server via MPI. As soon as both connections were made the testrun started. The testrun ended once the simulation-side finished sending data.

[4]https://apps.fz-juelich.de/jsc/hps/jureca/quickintro.html#interactive-sessions

- As described in the implementation of the simulation client, simulation runs of NEST were done in advance and written to file. This had two advantages. First, it assured an easy connection and sending of streaming data to the server. Second, the simulation output was reproducible and thus the measurements on throughput comparable. During a testrun, the simulation was simply reading data from file and sending every N lines to the server.
- Once the data was sent to analysis (and received by it), the data transfer was defined as complete. The data received on analysis-side was not further processed.
- The first measurement taken was the overall throughput of the workflow. The throughput was determined by counting the number of packages received from simulation and the number of packages sent to analysis.
- The second measurement was the percentage runtime of important steps of the transposition algorithms (e.g. initializing memory, copying data, etc.).

The parameters during testing:

- The number of processes for the parallel transposition. Due to the implementation of a single, shared memory buffer, this was limited to one compute node with 24 processes.
- Simulation, analysis and server were run on different compute nodes to test the in transit coupling and to avoid internal optimizations from MPI.
- The size of the buffer was varied, tests were done with a smaller and a larger buffer size.
- The amount of data from simulation and the amount of data requested by analysis was fixed for the experiments. The simulation consisted of a population of 1000 neurons and had approximately 47 million events sent in packages of $N = 10,000$ events. The analysis requested the data from all 1000 neurons.

The following table 5.1 shows an overview of the experiments:

	Experiment 1	**Experiment 2**
Simulation	1000 neurons, ~47 million events	
Analysis	request of 1000 neurons	
Package size	10000 events	
Number of processes	2,4,...,24	2,4,...,24
Buffer size	100 packages	10 packages

Table 5.1.: Parameters of the experiments. Both experiments have been conducted with both algorithms and 10 repetitions.

6. Results

In this chapter, the results of the implementation and the experiment conducted are presented. First, the implementation of the framework will be validated. Second, the results of the experiment on the transposition algorithms are shown and evaluated.

6.1. General in transit coupling validation

This section provides an analysis of the implemented modules and algorithms and if they meet the requirements from section 4.2. The results of the validation tests are presented and issues that occurred during testing are listed.

6.1.1. Requirement analysis

The technical requirements T1-T6 could be fulfilled. T1 (Databuffer) is fulfilled with the implementation of the ring buffer. Requirements T2-T6 are addressed in the next section, which presents the results of the experiment. First, the use of shared memory and MPI one-sided communication automatically fulfills T2 (Arbitrary data access). Second, the tests show, that the implementation of the two algorithms fulfills T3-T5. Finally, T6 (Send to analysis) is used as measurement during the experiment.

The results of the experiment show that the implementation also meets the requirement U1 (Fast data transfer). The requirement U2 (Flexible data transfer) is fulfilled in conjunction with U4 (Exchange of parameters). The successful exchange of parameters before simulation starts, allows for a flexible steering of data transfer. Requirement U3 is not yet met. All testing and the experiment was done to find an upper bound on the amount of data transferred. Therefore, flexible data access can be implemented in the future, by using the results from the work done in this thesis. Requirement U5 is fulfilled by the server. It receives the end-of-simulation signal and forwards it to analysis.

6.1.2. Implementation analysis

Module `simulation.py` For the purpose of proving the concept of in transit coupling in this thesis, the simulation of a simulation was sufficient. The output of a NEST simulation was read from file and sent in a data stream of fixed size packages. Even though this may not be realistic for actual simulations, it has

advantages. The size of the simulation, i.e. number of events per second, can easily be changed by varying the package size. This allowed for comparable and more precise tests and experiments on speed and throughput.

Module `analysis.py` The module could receive the data sent by the server and thus complete the in transit coupling. This was tested with different settings on server-side, i.e. number of processes and number of neurons. The use cases of Elephant were tested with the preliminary visualization designs shown in chapter 3.

Module `server.py` The client-server connections worked as described in section 5.1. This is proven by the exchange of parameters beforehand and by the successful data transfer.

Module `data_handler.py` All functions of this module worked as intended. The module contains the main functionalities, namely data reception, transposition, exchange, conversion and sending. Their validation is described in the next section 6.1.3. For all validation tests, the reproducibility of the simulated simulation could be utilized.

6.1.3. In transit workflow analysis

data reception The successful data reception depends on three parts of the implementation. First, a working connection with MPI, i.e. the intercommunicator between simulation and server. Second, the memory to receive the data into. Third, a working buffer logic. The following tests were done for validation: 1) Counting the number of packages sent by the simulation client and received by the server. 2) Tracking the head of the ring buffer. 3) Comparing the sent and received data for completeness and correctness. 4) Sending and receiving the end-of-simulation tags.

data transposition The validation of the data transposition algorithms could be tested separately. Examples, as shown in figure 4.1, were used to confirm the correctness.

data exchange For the validation of the correct data exchange as described in section 5.5, several testruns with a different number of processes were done. The results, the spiketimes of the neurons which arrived at analysis-side, were compared for differences. The expected result could be confirmed: no differences were observed during the tests. An additional test for the correct

communication scheme was done by counting the number of MPI `gatherv` operations.

data conversion For the use case of this thesis, the data conversion consisted of sending the transposed data in form of `numpy arrays`. This was done automatically due to the use of `numpy` in the transposition of the data and the `mpi4py` module. A modular functionality which supports different data types and formats is still needed for future applications.

data sending The basic use of the `analysis.py` module (see above) did validation tests for correct data transfer. However, for the experimental setup the data transfer was done with non-blocking communication in MPI, in order to avoid limitations of the receiving side. This caused some issues with the available MPI implementations on the JURECA supercomputer. One implementation of MPI was missing the Client-Server functionalities, while the others resulted in errors when using the non-blocking `irecv` call in mpi4py. As a result, the module did not properly receive and further process the data during the experiments.

6.2. Experiments: Parallel data transposition

This section presents the results of the experiment described in section 5.6. The first part shows the results of the tests and comparison of the two implementations of the parallel data transposition, namely the sort and append algorithm described in section 5.4. In the second part, the results of the tests on buffer size are shown and the impact of the buffer size on the data transfer is described.

6.2.1. Definition of speedup for the experiments

The tests performed for the experiments included an increase in the number of processes for data transposition. However, the design of the framework and the setup of the experiment do not enable classic scaling and speedup tests. This is due to the fact that the duration of the simulation is fixed and each testrun ends as soon as the last package from simulation arrives at server-side. Since the classic weak and strong scaling tests measure the runtime of a program, the following variation is defined and used for comparison:

Definition Speedup: The speedup of the program is defined by measuring the number of packages sent to analysis with an increasing number of processes. First, a sequential data throughput (D_s) and a parallel data throughput (D_p) are

defined. The sequential data throughput is obtained by measuring the transfer rate without parallelization. Here, this means running the in transit server with two processes, one for receiving data from simulation and one for (sequential) processing and sending to analysis. Finally the speedup is defined as: $S(p) = \frac{D_s}{D_p(p)}$, where p is the number of processes.

6.2.2. Comparison of the algorithms

With this definition of speedup, the results of the experiments were evaluated. Figure 6.1 shows the speedup for both algorithms, with the transfer rate (the rate between packages sent and received) on the right y-axis. The optimal speedup is

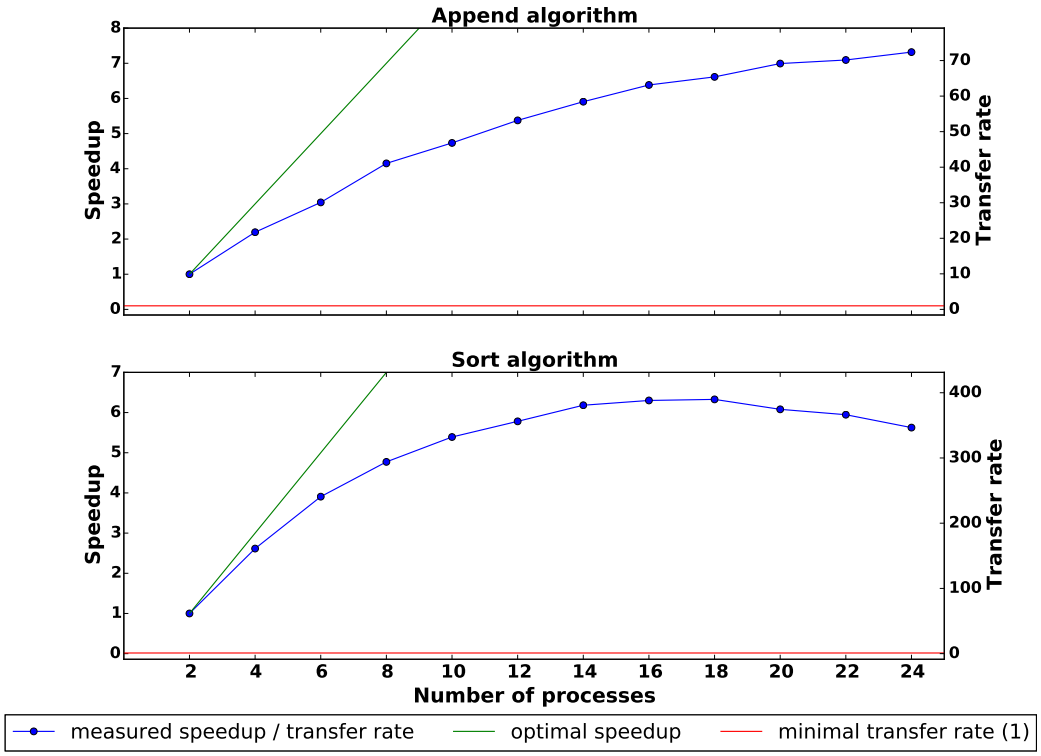

Figure 6.1.: Speedup of the algorithms, as defined in this experiment. There are two apparent differences: 1) The sort-algorithm processes $5-6$ times more data 2) The append algorithm shows better scaling behavior since it does not decline.

shown by the green line, the minimum required transfer rate before losing data (1:1 ratio between received and sent packages) is indicated by the red line. There are two major differences to notice. First, the transfer rate of the sort-algorithm is five to six

times larger than the transfer rate of the append-algorithm. Second, the speedup curves show differences in the behavior of the algorithms for a larger number of processes. Toward reaching the maximum number of processes per node (24), the speedup of the append algorithm seems to approach a limit but still grows, whereas the speedup of the sort algorithm clearly reaches a maximum and declines.

An early decision for one of the algorithms, judging only by the amount of data transferred, would favor the sort-algorithm. However, one of the user requirements (U1), is fulfilled by not losing any data received by simulation. This requirement is easily fulfilled by both algorithms. So, on the one hand, both algorithms meet the requirement of a minimum transfer rate, i.e. they do not lose packages. On the other hand, as discussed in section 4.1.2, duplicate data can be desirable in some use cases.

The huge difference in the number of packages that both algorithms transferred, can be attributed to a large buffer. A large buffer means more data for each process and thus more memory allocations for the append algorithm. This correlation is further investigated in the experiment in section 6.2.3.

Overall, the plots show that both algorithms do not scale well due to the communication overhead caused by the data exchange (see section 5.5). This underlines the need for the mentioned optimization of the data exchange problem. However, the append algorithm shows better scaling behavior (the speedup does not decline).

For further comparison, the percentages of the total runtime of important steps in both algorithms were measured. Especially computational steps and steps that require communication. Figure 6.2 shows the results of this measurement.

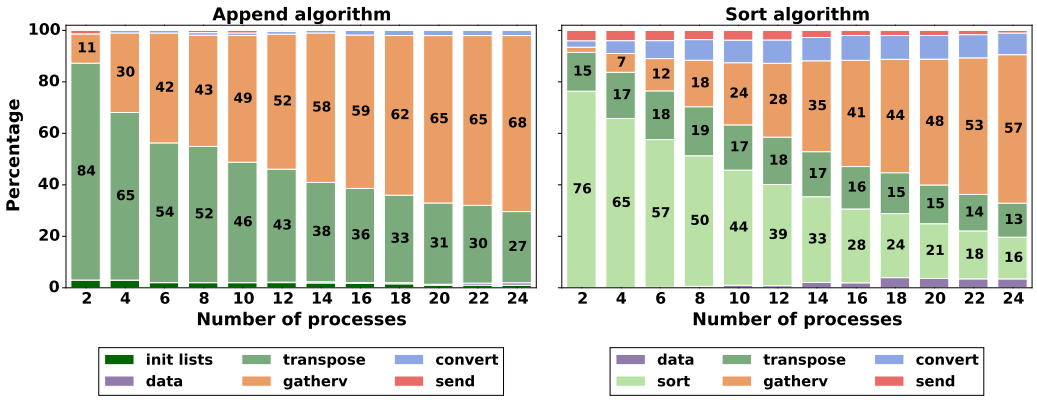

Figure 6.2.: Percentages of the total runtime of algorithmic steps. Communication is labeled in orange, computations are labeled in green. An increase in communication and decrease in computations is noticeable.

The communication, i.e. the `gather` operations, is labeled in orange. Algorithmic computations, such as sorting or index calculations, are labeled in green. Both algorithms show an percentage increase in communication and percentage decrease in computations. This result is expected, since an increase in the number of processes means an increase in the amount of communication.

Usually, a shift from more computation to more communication is noticeable in the speedup in a parallel program. This fact can be evaluated in conjunction with the result shown in figure 6.1. Figure 6.3 shows the parallel efficiency and the percentage computations of both algorithms. Parallel efficiency is defined as $E(p) = \frac{S(p)}{p}$, the ratio between speedup and number of processes. The ideal parallel efficiency would be 1 for all number of processes.

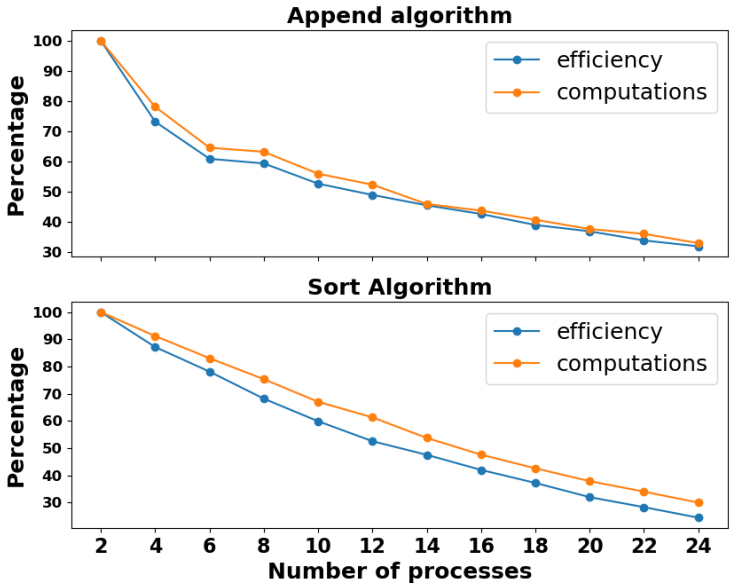

Figure 6.3.: The correlation between efficiency and percentage of computations of both algorithms. For comparison reasons efficiency is scaled to percentage.

The poor scaling of both algorithms and therefore the declining efficiency is strongly correlated with the decreasing percentage of computations done. This strong correlation fits the above assumption, that increasing communication is the main cause of a declining data transfer rate.

6.2.3. Impact of buffer size on data transfer

The tests with varying buffer size were done with the same settings as the previous tests. The buffer size chosen for this experiment was ten times smaller than before, i.e. it could store 10 packages from simulation instead of 100. Therefore, the algorithms only had to transpose and exchange a tenth of the previous amount of data for each access to the buffer. However, an exact increase of factor ten is not expected, due to the additional communication needed.

Figure 6.4 shows the result of this experiment compared to the results from before. The transfer rate is used for comparison, since a speedup comparison of two different experiments is not possible (different sequential data throughput).

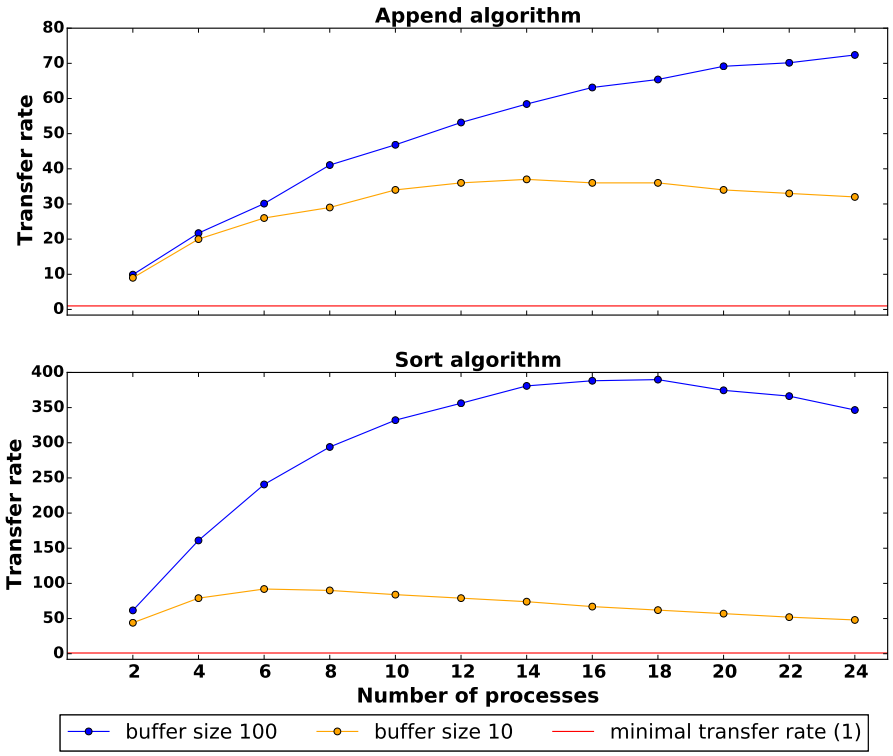

Figure 6.4.: Transfer rate of the algorithms compared (larger and a smaller buffer size). Both algorithms show a significant drop in transfer rate for the small buffer size. Both still meet the minimal required transfer rate before losing data.

The ideal algorithmic behavior would be to transfer the same amount of data, independent of the buffer size. Both algorithms still meet the minimal required

transfer rate before losing data. Also, the sort algorithm still performs better overall, but with only 2-3 times the amount of data transferred. However, the plots show a noticeable drop in transfer rate, especially for the sort algorithm.

This difference in behavior between append and sort algorithm becomes even more evident, when comparing the efficiency and decrease in computations (see figure 6.5), as done before for the larger buffer size. The decrease in computations

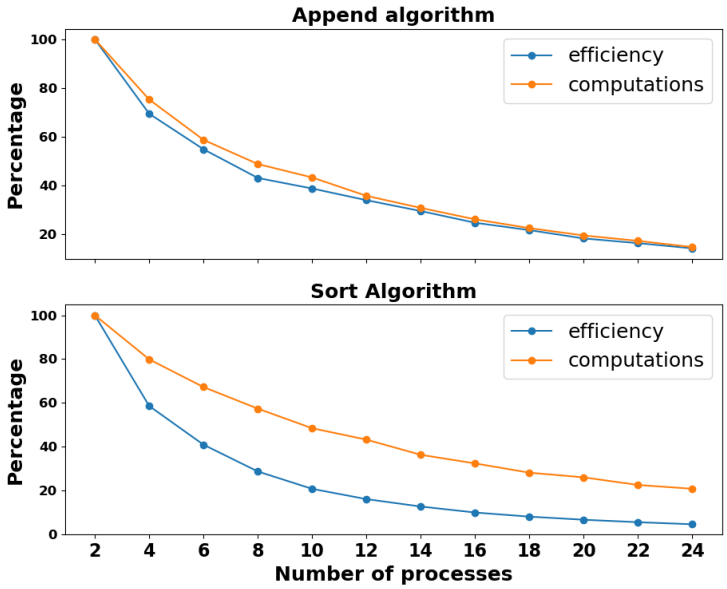

Figure 6.5.: The correlation between efficiency and percentage of computations for a smaller buffer size. The increase in communication has more impact on the efficiency of the sort algorithm.

and therefore increase in communication has clearly more impact on the efficiency of the sort algorithm as shown by the larger gap between efficiency and computation in figure 6.5. This is mostly explained by the increase in number buffer accesses (10 times more for the smaller buffer) and therefore increase in communication.

The considerably better scaling of the append algorithm with smaller buffer size can be explained, by comparing the result of this experiment with the results of the first experiment with buffer size 100. The algorithmic advantage of the sort algorithm is lower with smaller amounts of data per buffer access and more frequent communication. While communication also impedes the append algorithm, the actual `append` operation takes advantage of smaller packages sizes, due to over allocation[1].

[1]Source code, line 52 ff: https://github.com/python/cpython/blob/master/Objects/listobject.c

This effect should be more noticeable, for even smaller buffer sizes, i.e. the amount of data to append. This result can be used as tendency, even though further testing on this effect has to be done.

In summary, the sort algorithm performs better and should be the choice for the current implementation of the framework. However, the tendency towards the append algorithm for smaller amounts of data should not be neglected and further experiments and tests need to be done in the future.

7. Conclusion and Outlook

This thesis describes the development and implementation of an in transit coupling framework for a use case consisting of a neuroscientific simulation and analysis in HPC. The framework is able to transfer streaming data between workflow tasks, that are running on separate nodes of a supercomputer, without disk I/O operations in between.

In the first step of the development, scientific use cases have been chosen and formulated in collaboration with users and developers of the analysis toolkit Elephant. These use cases provide requirements for the development of the framework from an end users' perspective. Some of the requirements are specific to the scientific field of neuroscience and involve settings or parameter exchange. Others have impact on technical details in the implementation, such as data transfer rate. The design of the framework is guided by these scientific use cases and their requirements. Furthermore, four key concepts have been identified and described. They contain specifications for the design on the receiving, processing and sending of data.

Two of the key concepts are critical to the use case of this thesis and have therefore been further investigated. The first concept describes the data translation between the output and input of two subsequent tasks. In this case the transposition of spike events of neurons. The second key concept is a direct consequence of the parallelization of the transposition and involves communication between processes, i.e. exchange of data. The investigation of the second concept has led to the implementation of the data exchange with MPI gather operations. This data exchange algorithm has successfully been tested for correctness.

Two different algorithms for the key concept of data transposition have been implemented and an experiment for comparison has been conducted. The experiment measures the performance of both algorithms for an increasing number of processes and different amounts of data. The first algorithm (*append*) makes heavier use of memory allocations, while the second algorithm (*sort*) is computationally more expensive. The results of the experiment show, that overall the sort algorithm has better performance. It can transfer 2-6 times more data depending on the settings of the experiment. The results also show that the append algorithm has better scaling with reduced amounts of data which could be important for future applications.

The framework has been implemented in the Python programming language and

tested on the JURECA supercomputer at the Forschungszentrum Jülich. It consists of several modules, that use a client-server model to connect to each other. It can exchange parameters and receive data. All implemented modules of the framework have successfully been validated for their functionality. All but one requirement from section 4.2 have been met.

There are many opportunities for future work in form of extensions, optimizations and integration of the framework:

Data transposition algorithms and data exchange. Both developed and tested algorithms in this thesis successfully fulfilled the requirements of speed and accuracy. However, the behavior of both algorithms in different settings, such as increased data transfer from extreme-scale simulations or varying memory availability, needs to be evaluated. Additionally, optimizations to the algorithms or completely new algorithms for data transposition can be investigated and developed. Especially the increasing communication expenditure of the data exchange (section 5.5) will be a limiting factor when moving towards larger scales.

Data loss and data reduction. The advantage of asynchronous data transfer is the flexibility for user requirements, independent of the scale of the simulation. However, as discussed in section 4.1.2, this is at the risk of losing data. Instead of optimizing algorithms and communications, the possibilities and implications of data reduction of data sufficiency could be investigated. Work in this topic has been done for example with in situ filtering and aggregation in Bennet et al [5] or with data reduction combined with offline analysis in Foster et al [13].

Buffer and Memory Extensions. The current implementation of the framework makes use of MPI shared memory and a ringbuffer logic (section 5.3). A single data stream from simulation is received into a single shared memory buffer. Both sides of this communication could be extended to multiple senders and receivers in the future. The extension to multiple receivers has two general options. First multiple buffers on one node second multiple buffers on multiple nodes. While the first option is a straightforward extension, it would still limit the in transit coupling to be performed on one node due to the use of shared memory. The second option enables higher scaling of the coupling mechanism, but would include an additional layer of communication between nodes.

Developer Usability. Additional work to improve the usability of the framework in the future has been briefly mentioned in section 4.1. It is an important aspect for general extensions and applications of the framework but should be carefully managed. For example, the integration of third party libraries can be a powerful

tool and provide additional usability, but at the same time adds dependencies to the framework.

Simulation and analysis modularity. Even though the framework has been specifically designed and implemented for NEST and Elephant, the foundation for general extensibility for other neural simulators and analyses is provided. There are several aspects to this extensibility for future work. First, the connection to a simulation which is running on a supercomputer has to be defined and tested on different scales. Currently, connection to the simulation is made via the MPI Client-Server paradigm and the simulation itself is simulated (section 5.2). Second, for modular use of other (neural) simulators and analyses in the future, data models, schemes, and conventions need to be established. Finally, the integration of this framework in a broader workflow environment. For example there has been research and development on connectivity generation of neural networks [17] or interactive visualization and steering thereof [28].

Interactive Supercomputing. As demonstrated in [28] and also by the use cases in this thesis (chapter 3), interactive visualization and exploratory data analysis are important tools in science. Therefore, future work on the framework of this thesis would include extending the in transit coupling to live visualization and interactive steering on HPC systems.

Bibliography

[1] H. Abbasi, J. Lofstead, F. Zheng, K. Schwan, M. Wolf, and S. Klasky. "Extending I/O through high performance data services". In: *2009 IEEE International Conference on Cluster Computing and Workshops*. Aug. 2009, pp. 1–10. DOI: 10.1109/CLUSTR.2009.5289167.

[2] N. A. Akar, B. Cumming, V. Karakasis, A. Küsters, W. Klijn, A. Peyser, and S. Yates. "Arbor — A Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance Computing Architectures". In: 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Pavia (Italy), 13 Feb 2019 - 15 Feb 2019. IEEE, Feb. 13, 2019, pp. 274–282. ISBN: 978-1-7281-1644-0. DOI: 10.1109/EMPDP.2019.8671560. URL: http://juser.fz-juelich.de/record/861612.

[3] M. F. Aktas, G. Haldeman, and M. Parashar. "Flexible Scheduling and Control of Bandwidth and In-transit Services for End-to-End Application Workflows". In: *2014 Fourth International Workshop on Network-Aware Data Management*. Nov. 2014, pp. 28–31. DOI: 10.1109/NDM.2014.9.

[4] A. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel. "In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms". In: *Computer Graphics Forum* 35 (June 2016), pp. 577–597. DOI: 10.1111/cgf.12930.

[5] J. C. Bennett, H. Abbasi, P. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2012, pp. 1–9. DOI: 10.1109/SC.2012.31.

[6] J. C. Bennett, H. Abbasi, P. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. *Combining in-situ and in-transit processing to enable extreme-scale scientific analysis*. 2015. URL: https://www.osti.gov/servlets/purl/1248604 (visited on 07/2019).

[7] J. Bower and D. Beeman. *The Book of GENESIS*. Jan. 1998.

[8] DEEP Project - Jülich Supercomputing Centre. *Towards a Modular Supercomputing Architecture for Exascale*. 2013. URL: https://www.deep-projects.eu/ (visited on 07/2019).

[9] P. T. Dorian Krause. "JURECA: Modular supercomputer at Jülich Supercomputing Centre". In: *Journal of large-scale research facilities* 4 (2018). DOI: 10.17815/jlsrf-4-121-1.

[10] M. Dorier, M. Dreher, T. Peterka, J. M. Wozniak, G. Antoniu, and B. Raffin. "Lessons Learned from Building In Situ Coupling Frameworks". In: *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, ISAV 2015, Austin, TX, USA, November 15-20, 2015*. 2015, pp. 19–24. DOI: 10.1145/2828612.2828622. URL: https://doi.org/10.1145/2828612.2828622.

[11] Elephant, RRID:SCR_003833. *Elephant - Electrophysiology Analysis Toolkit*. 2015. URL: http://www.python-elephant.org/ (visited on 07/2019).

[12] Forschungszentrum Jülich GmbH - Jülich Supercomputing Centre. *Simulation Laboratory Neuroscience*. 2019. URL: https://www.fz-juelich.de/ias/jsc/EN/Expertise/SimLab/slns/_node.html (visited on 07/2019).

[13] I. Foster, M. Ainsworth, B. Allen, B. Julie, F. Cappello, J. Y. Choi, E. Constantinescu, P. E Davis, S. Di, W. Di, H. Guo, S. Klasky, T. Kurc, Q. Liu, A. Malik, K. Mehta, K. Mueller, T. Munson, G. Ostouchov, and S. Yoo. "Computing Just What You Need: Online Data Analysis and Reduction at Extreme Scales". In: Aug. 2017, pp. 3–19. DOI: 10.1007/978-3-319-64203-1_1.

[14] M. Gewaltig and M. Diesmann. "NEST (NEural Simulation Tool)". In: *Scholarpedia* 2.4 (2007). revision #130182, p. 1430. DOI: 10.4249/scholarpedia.1430.

[15] D. Goodman and R. Brette. "The Brian simulator". In: *Frontiers in Neuroscience* 3 (2009), p. 26. ISSN: 1662-453X. DOI: 10.3389/neuro.01.026.2009. URL: https://www.frontiersin.org/article/10.3389/neuro.01.026.2009.

[16] Helmholtz Portfolio Theme. *Supercomputing and Modeling for the Human Brain*. 2017. URL: https://www.fz-juelich.de/smhb/EN/Home/home_node.html (visited on 07/2019).

[17] P. Herbers. "Generating Neural network Connectivity from a Visual Representation". Masterarbeit. Ruhr-Universität Bochum, Nov. 2017.

[18] M. L. Hines and N. T. Carnevale. “The NEURON Simulation Environment”. In: *Neural Comput.* 9.6 (Aug. 1997), pp. 1179–1209. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.6.1179. URL: http://dx.doi.org/10.1162/neco.1997.9.6.1179.

[19] Human Brain Project. *Human Brain Project*. 2017. URL: https://www.humanbrainproject.eu/en/ (visited on 07/2019).

[20] T. Ichimura, K. Fujita, S. Tanaka, M. Hori, M. Lalith, Y. Shizawa, and H. Kobayashi. “Physics-Based Urban Earthquake Simulation Enhanced by 10.7 BlnDOF × 30 K Time-Step Unstructured FE Non-Linear Seismic Wave Simulation”. In: *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2014, pp. 15–26. DOI: 10.1109/SC.2014.7.

[21] *In situ Terminology Project*. 2019. URL: https://ix.cs.uoregon.edu/~hank/insituterminology/index.cgi (visited on 07/2019).

[22] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, and H. Abbasi. “Using Cross-layer Adaptations for Dynamic Data Management in Large Scale Coupled Scientific Workflows”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’13. Denver, Colorado: ACM, 2013, 74:1–74:12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503301. URL: http://doi.acm.org/10.1145/2503210.2503301.

[23] P. Johnsen, M. Straka, M. Shapiro, A. Norton, and T. Galarneau. “Petascale WRF simulation of hurricane sandy: Deployment of NCSA’s cray XE6 blue waters”. In: *SC ’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2013, pp. 1–7. DOI: 10.1145/2503210.2503231.

[24] J. Jordan, T. Ippen, M. Helias, I. Kitayama, M. Sato, J. Igarashi, M. Diesmann, and S. Kunkel. “Extremely Scalable Spiking Neuronal Network Simulation Code: From Laptops to Exascale Computers”. In: *Frontiers in Neuroinformatics* 12 (2018), p. 2. ISSN: 1662-5196. DOI: 10.3389/fninf.2018.00002. URL: https://www.frontiersin.org/article/10.3389/fninf.2018.00002.

[25] J. Jordan, H. Mørk, S. B. Vennemo, D. Terhorst, A. Peyser, T. Ippen, R. Deepu, J. M. Eppler, A. van Meegen, S. Kunkel, A. Sinha, T. Fardet, S. Diaz, A. Morrison, W. Schenck, D. Dahmen, J. Pronold, J. Stapmanns, G. Trensch, S. Spreizer, J. Mitchell, S. Graber, J. Senk, C. Linssen, J. Hahne,

A. Serenko, D. Naoumenko, E. Thomson, I. Kitayama, S. Berns, and H. E. Plesser. *NEST 2.18.0*. June 2019. DOI: 10.5281/zenodo.2605422. URL: https://doi.org/10.5281/zenodo.2605422.

[26] J. Kress, S. Klasky, N. Podhorszki, J. Choi, H. Childs, and D. Pugmire. "Loosely Coupled In Situ Visualization: A Perspective on Why It's Here to Stay". In: *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ISAV2015. Austin, TX, USA: ACM, 2015, pp. 1–6. ISBN: 978-1-4503-4003-8. DOI: 10.1145/2828612.2828623. URL: http://doi.acm.org/10.1145/2828612.2828623.

[27] T. Marrinan, S. Rizzi, J. Insley, B. Toonen, W. Allcock, and M. E. Papka. "Transferring Data from High-Performance Simulations to Extreme Scale Analysis Applications in Real-Time". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018, pp. 1214–1220. DOI: 10.1109/IPDPSW.2018.00188.

[28] C. Nowke, S. Diaz-Pier, B. Weyers, B. Hentschel, A. Morrison, T. W. Kuhlen, and A. Peyser. "Toward Rigorous Parameterization of Underconstrained Neural Network Models Through Interactive Visualization and Steering of Connectivity Generation". In: *Frontiers in Neuroinformatics* 12 (2018), p. 32. ISSN: 1662-5196. DOI: 10.3389/fninf.2018.00032. URL: https://www.frontiersin.org/article/10.3389/fninf.2018.00032.

[29] T. Proix, F. Bartolomei, M. Guye, and V. K. Jirsa. "Individual brain structure and modelling predict seizure propagation". In: *Brain* 140.3 (Feb. 2017), pp. 641–654. ISSN: 0006-8950. DOI: 10.1093/brain/awx004. eprint: http://oup.prod.sis.lan/brain/article-pdf/140/3/641/24174967/awx004.pdf. URL: https://doi.org/10.1093/brain/awx004.

[30] T. Terraz, B. Raffin, A. Ribes, and Y. Fournier. "In Situ Statistical Analysis for Parametric Studies". In: *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*. Nov. 2016, pp. 35–39. DOI: 10.1109/ISAV.2016.012.

[31] R. A. Tikidji-Hamburyan, V. Narayana, Z. Bozkus, and T. A. El-Ghazawi. "Software for Brain Network Simulations: A Comparative Study". In: *Frontiers in Neuroinformatics* 11 (2017), p. 46. ISSN: 1662-5196. DOI: 10.3389/fninf.2017.00046. URL: https://www.frontiersin.org/article/10.3389/fninf.2017.00046.

[32] W. Usher, S. Rizzi, I. Wald, J. Amstutz, J. Insley, V. Vishwanath, N. Ferrier, M. E. Papka, and V. Pascucci. "libIS: A Lightweight Library for Flexible in Transit Visualization". In: *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ISAV '18. Dallas, Texas: ACM, 2018, pp. 33–38. ISBN: 978-1-4503-6579-6. DOI: 10.1145/3281464.3281466. URL: http://doi.acm.org/10.1145/3281464.3281466.

[33] V. Vishwanath, M. Hereld, and M. Papka. "Toward simulation-time data analysis and I/O acceleration on leadership-class systems". In: (Oct. 2011). DOI: 10.1109/LDAV.2011.6092178.

[34] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi. "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. May 2012, pp. 1352–1363. DOI: 10.1109/IPDPS.2012.122.

A. Configuration JURECA

Hardware Characteristics of the Cluster Module[1];

- 1872 compute nodes
  - Two Intel Xeon E5-2680 v3 Haswell CPUs per node
  - 75 compute nodes equipped with two NVIDIA K80 GPUs (four visible devices per node)
  - DDR4 memory technology (2133 MHz)
- 12 visualization nodes
  - Two Intel Xeon E5-2680 v3 Haswell CPUs per node
  - Two NVIDIA K40 GPUs per node
  - 10 nodes with 512 GiB memory
  - 2 nodes with 1024 GiB memory
- Login nodes with 256 GiB memory per node
- 45,216 CPU cores
- 1.8 (CPU) + 0.44 (GPU) Petaflop per second peak performance
- Based on the T-Platforms V-class server architecture
- Mellanox EDR InfiniBand high-speed network with non-blocking fat tree topology
- 100 GiB per second storage connection to JUST

[1] https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/Configuration/Configuration_node.html
Last visited on August 2019