## Title

Proximity Portability and in Transit, M-to-N Data Partitioning and Movement in SENSEI



## ISBN

9783030816261

## Authors

Bethel, E Wes
Loring, Burlen
Ayachit, Utkarsh
et al.

## Publication Date

2022

## DOI

10.1007/978-3-030-81627-8_20

Peer reviewed

# Proximity Portability and *In Transit*, M-to-N Data Partitioning and Movement in SENSEI

E. Wes Bethel[1], Burlen Loring[1], Utkarsh Ayachit[4], Earl P. N. Duque[5], Nicola Ferrier[2], Joseph Insley[2], Junmin Gu[1], James Kress[3], Patrick O'Leary[4], Dave Pugmire[3], Silvio Rizzi[2], David Thompson[4], Will Usher[6,7], Gunther H. Weber[1], Brad Whitlock[5], Matthew Wolf[3], Kesheng Wu[1]

**Abstract** In high performance parallel *in situ* processing, the term *in transit* processing refers to those configurations where data must be moved from $M$ ranks of a parallel data producer to $N$ ranks of a parallel data consumer. One of the central challenges in this setting is to determine a mapping of data from producer ranks to consumer ranks. This is a challenging problem for several reasons, such as when producer and consumer codes have different levels of concurrency, different scaling characteristics, or different data models. The resulting mapping and movement of data from $M$ to $N$ ranks can have a significant impact on aggregate application performance, particularly when the data consumer requires only a subset of the overall data for its task. This chapter focuses on the design considerations that underlie SENSEI's implementation to this challenging problem. These design considerations extend the core SENSEI architecture and include ideas like the need to accommodate flexibility in the choice of different partitioning methods, the ability for a data consumer to request and receive only the subset of data it needs for its particular operation, and the ability to leverage any of several different data transport tools. This idea of *proximity portability*, being able to use different data transport methods as part of an *in transit* workflow, is illustrated through the use of three different transport layers where switching from one transport tool to another is accomplished with only a configuration file change. The chapter also includes a performance analysis summary showing the performance gains that are possible, in terms of multiple metrics, such as memory footprint, time to solution, and amount of data moved, when using optimized partitioners in an *in transit* setting, gains that are made possible by the implementation shaped by specific design considerations.

[1]Lawrence Berkeley National Laboratory · [2]Argonne National Laboratory · [3]Oak Ridge National Laboratory · [4]Kitware, Inc. · [5]Intelligent Light · [6]University of Utah · [7]Intel Corporation

# 1 Introduction and Overview

In the regime of *in situ* processing, one of many particular configurations entails a scenario where data is moved across a network as it is produced to a separate application running on a separate set of hardware resources for analysis or visualization. In this scenario, data is produced on $M$ simulation ranks then consumed on $N$ consumer ranks, where typically $M >> N$. We refer to this configuration as *in transit* M-to-N processing, or more tersely as M-to-N processing, to capture both these concepts: that data is moved from producer to consumer, and that producer and consumer are running at different levels of concurrency.

In this configuration, a central challenge is the problem of M-to-N data redistribution from producer ranks to consumer ranks. Data redistribution refers to the process of determining how to map data from $M$ ranks to $N$ ranks, and also moving the data from one place to another, and doing so in a reasonably efficient and platform portable manner. One way the challenge arises is when producer and consumer run at markedly different levels of concurrency resulting in no clear and obvious mapping from one to another. Another is when producer and consumer each use a different data models, which would entail some level of data model reconciliation. Yet another way is when producer and consumer have vastly different scaling characteristics, which in turn lead to different levels of concurrency $M$ and $N$ for a given producer-consumer pairing on a given problem configuration. Finally, it is often the case that the consumer ranks do not require the complete problem domain from the producer, a situation that can occur during data reduction or subsetting operations, such as slicing or isocontouring.

This chapter focuses on design, implementation, and performance analysis issues of a general purpose solution to the M-to-N data redistribution problem encountered in *in transit* processing scenarios. The design considerations (Section 2) include topics related to SENSEI's adaptor architecture that focus on *in transit* scenarios, the central role of metadata, and how a partitioner uses metadata to compute a mapping from M-to-N ranks. These design principles allow SENSEI's implementation of the M-to-N *in transit* data redistribution methods to achieve *proximity portability*, whereby SENSEI-instrumented codes can make use of one of several different tools, such as HDF5, libIS, and ADIOS, for moving data between *in transit* processing stages with only a configuration file change (Section 3). A summary of an in-depth performance study of SENSEI's M-to-N *in transit* implementation at scale on a large HPC platform with multiple applications (Section 4) includes use of a full-scale physics code that uses adaptive mesh refinement (AMR), and analyzes performance across a number of metrics that include runtime performance, amount of data moved, time to solution, cost of solution, and memory footprint. The results show generality, broad applicability across a set of different data producers, data consumers, levels of concurrency, and varying partitioning algorithms.

## 2 Data and Execution Model Design Considerations for M-to-N, *In Transit* Processing

*In transit* processing is fundamentally different than *in situ* processing when considering how data is decomposed across the parallel ranks of producer and consumer. For example, consider an *in situ* configuration where a simulation code running at $M$-way parallel, which invokes *in situ* methods that are also run, by definition, at $M$-way concurrency. The simulation's data decomposition dictates what data is processed by each of the *in situ* ranks: the *in situ* method's data decomposition is imposed by the simulation's data decomposition. In contrast, in an *in transit* scenario, while an $M$-way parallel simulation code uses one data decomposition, the $N$-way parallel analysis code is likely to use a completely different problem and data decomposition. The central challenge is to determine how to map from one data decomposition to another in the M-to-N setting.

In this section, we discuss several design considerations that focus on different aspects of this complex problem: defining a mapping of data from M-to-N ranks, doing so in a way that can accommodate a variety of different producer/consumer code pairs, run at varying concurrency, and using a number of different potential mechanisms for moving data.

To begin, we present some background material about SENSEI's endpoint (Section 2.1) and adaptor design patterns (Section 2.2), which are foundational to the ability to swap in and out different *in situ* methods without having to recompile the simulation, or data producer, code. Then, we proceed to describe the metadata needed to describe the producer's data model to the *in transit* consumer ranks (Section 2.3). The metadata is input to the partitioner (Section 2.4), which is responsible for computing a mapping from $M$ producer to $N$ consumer ranks. These elements are brought together to enable data movement (Section 3), which includes the idea of *proximity portability*, or the use of multiple data transport tools with the ability to switch between tools at runtime simply by changes to a configuration file.

### 2.1 Endpoint

The *in transit* configurations we focus on are those consisting of multiple MPI parallel applications that run concurrently on HPC systems, where one parallel MPI job is a data producer and the other parallel MPI job is a data consumer. We use the term *endpoint* to refer to these parallel applications, which are compiled and linked with SENSEI and other related libraries, and that consume and process data. Note that in some circumstances, a data producer might be a data consumer, in which case it would be considered to be an endpoint as well. Figure 1 shows a illustrative *in transit* example, with the endpoint shown as Figure 1(f).

The challenge is to find a partitioning and to move data from a simulation with 5 blocks of data distributed on 5 MPI ranks to the endpoint which is running on 2 MPI ranks. The endpoint is universal in the sense that it may be configured at run time to

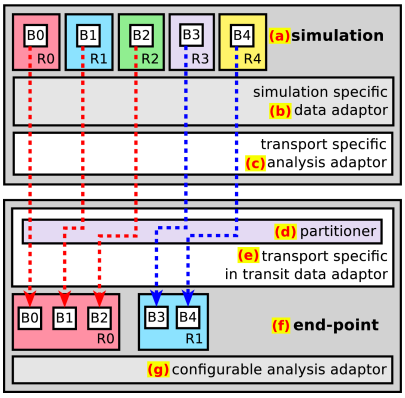



Fig. 1: SENSEI *In Transit* architecture. A transport comprised of a pair of adaptors moves data from the simulation running on $M$ MPI ranks to the end point running on $N$ MPI ranks. Data is processed by any of the usual SENSEI analyses, via the Configurable Analysis Adaptor. The system is comprised of the following components: (a) simulation, (b) simulation specific data adaptor, (c) transport specific analysis adaptor, (d) partitioner, (e) transport specific data adaptor, (f) endpoint, (g) configurable analysis adaptor. As shown on the left, the simulation has 5 blocks distributed on 5 ranks, and the partitioner has mapped these approximately evenly onto the end point's 2 ranks. This image adapted from our previous work [15].

receive and process data from any instrumented simulation without modifications to either the endpoint or the simulation. This is achieved via XML files provided on the command line, and any supported I/O or *in situ* data processing library may be used. Simulations wishing to run in an *in transit* configuration only need supply these two XML configurations.

## 2.2 Adaptor pattern

In our design there are only two actions: *invoke* and *fetch*. *In situ* processing — including analysis, visualization, and I/O — is periodically *invoked* by a simulation. In response to the *invoke* action, data processing code *fetches* data to process.

This design pattern is realized by two fundamental SENSEI adaptor types, the *analysis adaptor*, which is used to *invoke* processing, and the *data adaptor*, which is used to *fetch* the needed data. The adaptors define APIs that enable the invocation of a specific action without the need for the invoking code to know anything about the underlying implementation. The adaptor pattern allows for a change in the underlying implementation without the need to modify the code that invokes the adaptor methods.

The *data adaptor* is used to *fetch* data in both *in situ* and *in transit* configurations. Any consumer of data, whether it be for I/O, visualization, or analysis, makes use of the data adaptor to fetch data. Every simulation needs to provide a *simulation specific data adaptor* (Figure 1(b)) that is passed as a part of the *invocation* of *in situ* processing. Similarly the fetch operations of every I/O library are exposed to the system via an *I/O library specific data adaptor* (Figure 1(e)) that is passed as part of the invocation of processing. Through the use this adaptor design pattern, data processing codes that consume data need not be modified when run either in an *in situ* or an *in transit* configuration. A detailed figure illustrating these relationships appears in other publications (c.f., [15], Figure 4).

To cope with the additional complexities of the *in transit* configuration, I/O library specific data adaptors are derived from the *in transit data adaptor* (Figure 1(e)), which is itself derived from the `DataAdaptor` type. The *in transit data adaptor* adds, to the adaptor API, the APIs for use by the endpoint for management and control of connection and data movement from the simulation as well as APIs for interfacing with our partitioning mechanisms (Figure 1(d) and Section 2.4).

The *analysis adaptor* is used to *invoke* processing, both in *in situ* and *in transit* configurations. New analysis and I/O capabilities are exposed to simulators through the introduction of *transport specific analysis adaptors* (Figure 1(c)). The role of transport specific analysis adaptors is to initialize and configure an *in situ* or I/O library for some user-specified processing or movement. For example, in response to invocation by the simulation, the transport specific analysis adaptor will fetch and transform data, and then potentially provide that data to another library for processing or movement to the endpoint.

In both *in situ* and *in transit* configurations the simulation initiates the invocation. In the *in transit* configuration, this invocation initiates a data movement phase where data is transferred to the endpoint for processing. In the endpoint, an I/O library specific *in transit* data adaptor (Figure 1(e)) listens for the invocation, and forwards it into a library specific analysis adaptor.

The system implements runtime configurability through a process of runtime delegation. Configurable adaptor implementations create and initialize a library-specific adaptor instance from a user-provided XML configuration file (Figure 1(g) and SENSEI's Configurable Analysis Adaptor. Calls made to a configurable analysis adaptor are forwarded directly to the library-specific instance. Therefore, a simulation instrumented to use the configurable analysis adaptor gains access to all available I/O and *in situ* libraries. The endpoint gains access to all available I/O, data movement, and *in situ* processing capabilities through a single interface.

## 2.3 Metadata

In the context of *in situ* and *in transit* processing, the term *metadata* refers to "information about data" that is shared and exchanged between producer and consumer. This metadata includes values like the size and dimensionality of the mesh on the

sender side. It also includes much more detailed information that spans four different categories, including: data sets, arrays, data blocks, and where appropriate, AMR-specific information.

The role of metadata is central to managing data in support of M-to-N redistribution for *in transit* use scenarios. It is used to describe simulation data and its mapping onto the simulation's $M$ MPI ranks. Partitioners (Section 2.4) use the simulation metadata to compute the desired mapping of data onto the endpoint's $N$ MPI ranks. Data transports use metadata from multiple sources — the simulation and partitioner — to coordinate data movement between producer and consumer. The rich metadata object in SENSEI, which includes 34 different metadata values [15], enables transports to perform in-flight data transformations such as mapping one mesh onto another, and enables partitioners to implement a number of load balancing strategies.

## 2.4 Partitioner

Data partitioning in an M-to-N *in transit* scenario refers to the process of defining a mapping from an $M$-way parallel data producer to an $N$-way parallel data consumer. The process of partitioning may be straightforward, such as when $M == N$ in the case of traditional *in situ* processing, or it may be substantially more complex. Consider for example how some global parallel FFT implementations may require "pencil" or "slab" domain decomposition [17], or how two common approaches for parallelizing streamline computations — parallelize over blocks or parallelize over seeds — each require a different type of data partitioning [3]. Given the diversity of potential ways data may need to be transformed and redistributed in *in transit* use cases, one of our design objectives is to make it straightforward to use any of a number of different potential partitioning methods.

One implementation result of this design objective is that the partitioner is separate and distinct from both data producer and data consumer so that different partitioning methods may used with different combinations of producer and consumer. Similarly, the partitioner is separate and distinct from the data transport so that it is possible to use of any of a number of potential implementations of *in transit* data movement tools (Section 3).

A later section that examines the performance of *in transit* M-to-N configurations at scale (Section 4) makes use of two types of partitioners: a *default* partitioner and an *optimized* partitioner. A default partitioner will move the entire dataset regardless of how much is needed by the consumer. The default partitioner simply invokes a default block-based equipartitioning algorithm over the entire simulation domain. In contrast, *optimized partitioners* are provide only the subset of data actually needed by the consumer. These optimized partitioners use the metadata provided by the simulation to determine the minimum subset of data blocks needed for a particular operation.

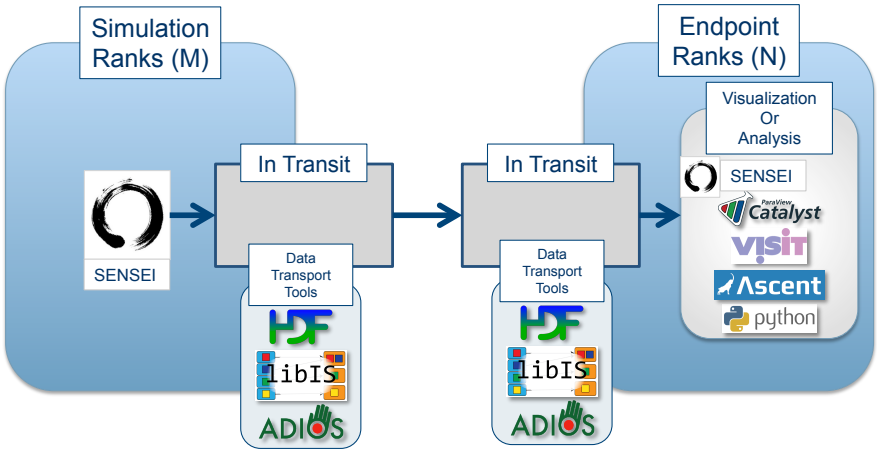



Fig. 2: An $M$-way parallel simulation sends data to an $N$-way parallel data consumer over one of several different potential data transport tools: HDF5, libIS, or ADIOS. Image courtesy W. Bethel and B. Loring.

For these two partitioner types, default and optimized, we will examine two use scenarios where only a subset of data is needed to complete the operation. The first is a slice extract, which takes a 2D slice from a 3D volume, and the second is an isosurface extract, which is computing an isosurface from a 3D volume. In the case of the slice extract, per-block bounding box metadata is tested for intersection with the slice plane. Only those blocks intersecting the plane are needed to compute the extract. In the case of the isosurface extract, per-block array range metadata are tested for intersection with the set of isocontouring values. Only those blocks where the data range brackets an an isocontouring value are needed to compute the extract. Blocks not needed in the calculation are not assigned to any rank and as a result are not moved to nor processed by the consumer. Once the needed set of blocks are identified the block-based equipartitioning algorithm assigns them to available endpoint ranks.

## 3 Proximity Portability and SENSEI's Use of Multiple Data Transport Tools

One of SENSEI's design objectives is to enable a "write once, run everywhere" approach. In this approach, the key idea is the addition of SENSEI instrumentation code to a data producer, like a numerical simulation, then enables use of multiple potential *in situ* endpoints and tools without the need for any further instrumentation code changes on the data producer side. In other words, once a simulation is instru-

mented with SENSEI, selection of a variety of different *in situ* tools is accomplished by changes to a configuration file.

This concept of *tool portability* extends to the notion of *in transit* processing as well, where a SENSEI-instrumented data producer code can run either *in situ* or *in transit* without any instrumentation changes, and may also take advantage of a growing collection of tools that perform data movement, a concept we refer to as *proximity portability.* This design objective is shown in Figure 2, where we have an $M$-way parallel simulation producing data that is sent over one of several potential transports to an $N$-way parallel endpoint that is performing visualization, analysis, or some other data-intensive operation. The subsections that follow present information about the SENSEI's *in transit* proximity portability through its use of three different mechanisms for moving data between producer and consumer: HDF5, ADIOS, and libIS.

### 3.1 HDF5 *In Transit* Data Transport

HDF5 is a mature parallel I/O library and file format that is widely deployed and used on HPC systems around the world [8] for projects ranging from parallel I/O for some of the world's largest computer codes, to use as a storage format for long-lived data from observations and experiments. Its prevalence as a parallel I/O library motivated recent work aimed at cultivating a better understanding of design and performance issues that would arise when leveraging HDF5 for *in transit* data staging and movement.

The work by Gu et al., 2019 [10] studied use of HDF5 for *in transit* data movement and staging, and specifically with a configuration that uses NVRAM Burst Buffers presented as a filesystem on an HPC platform. The resulting design and implementation is an HDF5-based mechanism for *in transit* data transport that is accessible to SENSEI-instrumented codes. This data transport capability for M-to-N, *in transit* processing is accessible as one of several potential *in transit* transport mechanisms that can be selected through an XML-based configuration file.

Figure 3 shows a block diagram of the SENSEI-HDF5 transport mechanism, where HDF5 in this case uses NVRAM-based Burst Buffers (BB) for data staging. Following the design patterns used by other data transport mechanisms in SENSEI, the SENSEI-HDF5 transport mechanism consists of two related adaptors, the *Analysis Adaptor* and the *Data Adaptor*. The Analysis Adaptor implements the SENSEI interface for outputting data from data producers, and the Data Adaptor implements the input interface for consumers to ingest data. As of the time of this writing, SENSEI uses the VTK data model as a bridge between Analysis and Data adaptors. Therefore, in the SENSEI-HDF5 transport, the Analysis Adaptor receives VTK data, and stores to HDF5, while the Data Adaptor reads from HDF5 and returns VTK data for SENSEI.

In our HDF5-based *in transit* data transport, we are able to leverage specialized hardware, such as BBs, for data staging. When the BB is presented as a filesystem

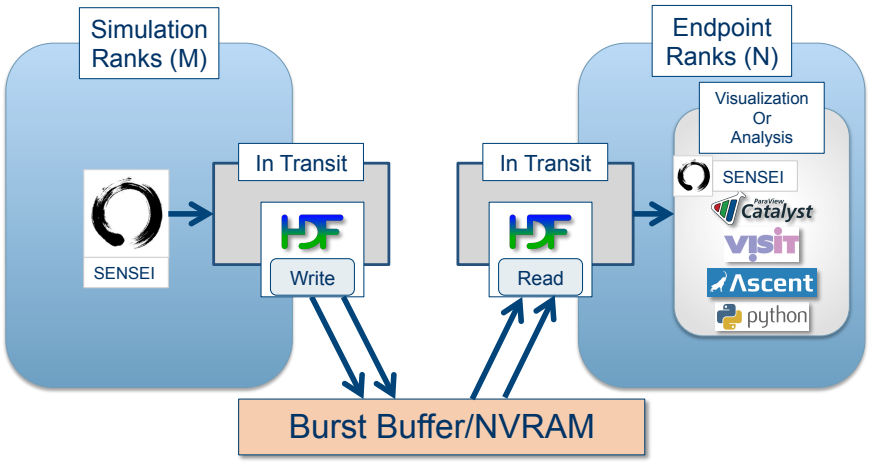



Fig. 3: The HDF5 transport layout. This particular illustration shows a configuration where Burst Buffers serve to stage data as it is moved from parallel producer to consumer. This image adapted from our previous work [10].

to users, then our HDF5-based transport can make use of the BB when the filename in the HDF5-SENSEI configuration file points to a location on the BB-resident filesystem. One benefit of using BBs for *in transit* data transport is that this staging mechanism may provide a larger maximum data footprint than is possible with conventional DRAM, memory-based methods. In other words, the total amount of BB on an HPC system often exceeds the total distributed memory footprint. When using a BB presented as a filesystem, one might expect that their use for data staging would be slower than a purely memory-based approach for staging, one that does not involve writes and reads to a BB-resident filesystem.

Recent studies [10] measure and compare the performance of the SENSEI-HDF5 *in transit* data transport over BB with one that uses a socket-based, memory-memory copy from one node to another. What is unexpected is that the BB configuration often completes the analysis use cases in less time than the socket-based transport mechanism as seen in both sets of tests reported in Figure 4. When compared to using a traditional disk-based filesystem for staging, both socket- and BB-based approaches are significantly faster, as is visible in the left image of Figure 4.

The right of Figure 4 shows only two data transport options: socket and BB, but with more test configurations than in the chart on the left. At the outset of this experiment, we expected the socket-based option to run more quickly than the BB configuration because it uses memory-memory transfers. However, these particular experiments show the opposite: the BB configurations run more quickly than the memory-memory configuration; this result was a surprise. The most likely reason for this performance difference is that HPC systems architects and developers tend to optimize for file-based I/O, as opposed to socket-based operations. In this case,

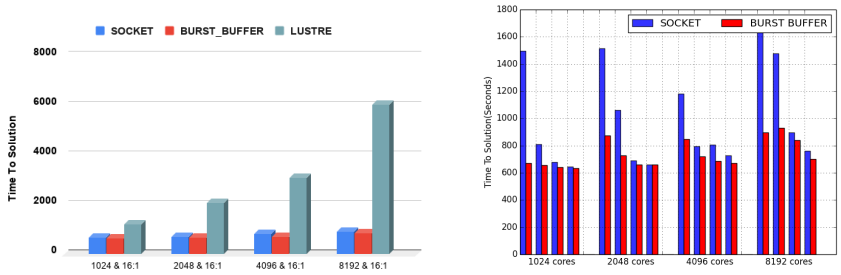



Fig. 4: Left: Time to solution for socket-based (ADIOS-FLEXPATH), NVRAM/Burst Buffer (HDF5), and disk (HDF5) approaches. Right: Time to solution for socket-based(ADIOS-FLEXPATH) and Burst Buffer (HDF5) approaches. These images are reprinted from our previous work [10].

```
<sensei>
  <analysis type="hdf5"
     filename="/burst/buffer/file"
     method="stream" enabled="1" />
</sensei>
```

(a) Configuration file for specifying HDF5 as the data transport mechanism for *in transit* data movement. Here, the filename path points to the NVRAM/Burst Buffer.

```
<sensei>
   <analysis type="adios1"
     filename="./test"
     method="FLEXPATH" enabled="1 "/>
</sensei>
```

(b) Configuration file for specifying ADIOS-FLEXPATH as the data transport for *in transit* data movement.

Fig. 5: Configuration files that show how to use HDF5 (left, (a)) or ADIOS-Flexpath (right, (b)). Changing from one transport to the other requires no coding changes, only a different configuration file.

the NVRAM/BB, file-based approach is faster than a socket-based approach due to system software architecture and its optimizations for file-based I/O.

In addition to runtime as a performance measure, this same work also examines other metrics, such as memory consumption. One interesting observation is that the memory footprint requirements of the memory-based staging approach in these tests are significantly larger than the memory footprint of the HDF5/BB-based approach. On the simulation side for the memory-based staging, data is buffered in memory pools, the size of which grows as a function of how quickly the data can be moved to the *in transit* method. In contrast, the HDF5/BB approach does not exhibit these same memory requirements. See Gu et al., 2019 [10] for more details.

Switching between data transports is a simple as changing a configuration file. For the test configurations in these studies, Figure 5 shows two configuration files: one for HDF5/BB, and the other for ADIOS-Flexpath. In the HDF5 configuration file, the pathname points to a location on the BB filesystem. In the ADIOS configuration file, the "method" of `FLEXPATH` requires a unique name, which in this case, is encoded into the `filename` field of the configuration file.

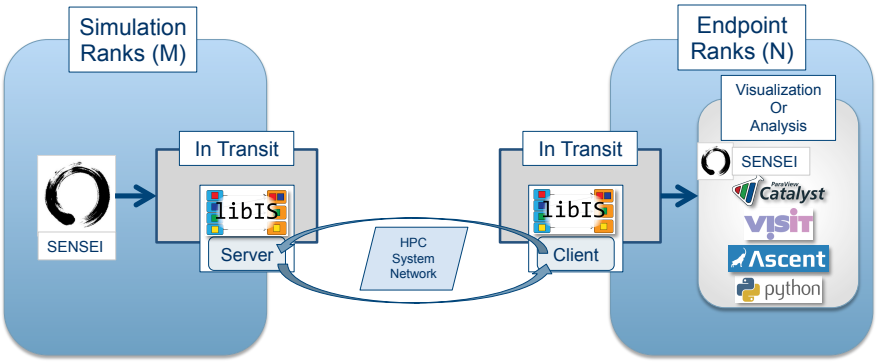



Fig. 6: libIS *in transit* architecture with SENSEI. Image courtesy S. Rizzi, N. Ferrier, and W. Bethel.

### 3.2 libIS *In Transit* Data Transport

libIS [19] is a lightweight library for *in transit* data transport. It uses a client-server model where clients (consumers) can request data from servers (producers) on an as-needed basis. libIS has been specifically designed for asynchronous, *in transit* analysis, where the simulation and analysis run decoupled from one another. The simulation-side library, coupled to the simulation either through our SENSEI interface or directly, exposes the simulation as a data server. The client-side library queries this server for new timesteps (Figure 6). Simulation and analysis can run on the same nodes, separate nodes, or within the same MPI launch command.

With *in transit* intrumentation, the simulation and visualization can each run in parallel and asychronously, or "loosely coupled." This allows for interactive *in situ* visualization without blocking the simulation. This configuration is in contrast to tightly-coupled interactive visualization approaches where the simulation blocks while the visualization method runs. Furthermore, by using libIS's connect and disconnect support, such interactive viewers can be used sporadically, e.g., to check in on a long running simulation, or to begin monitoring after some event without requiring that additional nodes be set aside for the entire run.

libIS has been demonstrated at scale on Theta (Argonne National Laboratory) and Stampede 2 (Texas Advanced Supercomputing Center) supercomputers [19]. Use cases include molecular dynamics simulations in LAMMPS instrumented with SENSEI and an interactive ray tracing viewer. At the time of this writing there is work in progress to integrate libIS as an additional transport in the SENSEI M:N data and execution model.

Usher et al., 2018 [20] explored interactive in transit visualization of molecular dynamics simulations, the visual results of which appear in Figure 7. They used the LAMMPS simulation code instrumented with SENSEI and the libIS *in transit* library to move simulation data over the network to a set of nodes running OSPRay, a high

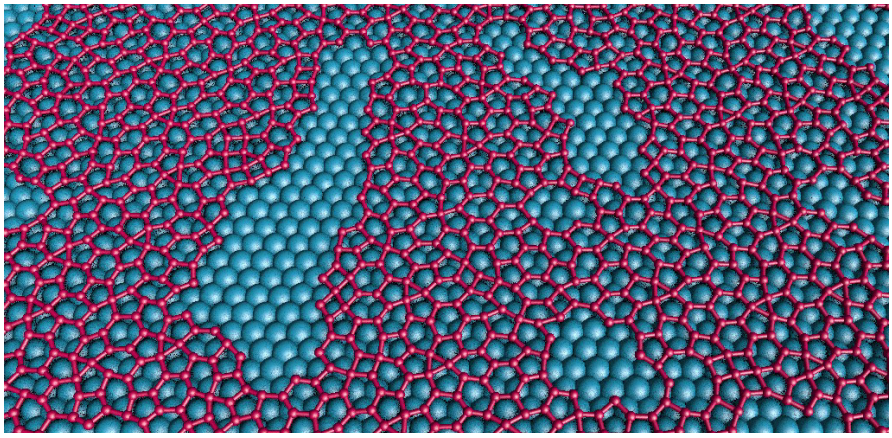


Fig. 7: Interactive *in transit* visualization of a 172k atom simulation of silicene formation with 128 LAMMPS ranks sending to 16 OSPRay renderer ranks, all executed on Theta in the `mpi-multi` configuration. This image is reprinted from our previous work [20].

performance ray tracer for CPUs [21]. The libIS performance study used a simulation input deck that is part of an active scientific research project at Argonne National Laboratory that performs atomic-level simulations of the formation of silicene [4].

In study's configuration, the analysis and render components both run on the same HPC resource. The render side uses libIS to query data from the simulation, while the OSPRay data-distributed API is used to ray trace the data. The renderer continues to query for data as the simulation runs, thus enabling the user to monitor the simulation state while it evolves over time. Along with rendering the data, the render side also supports running some local VTK pipelines to process the data, e.g., to compute bonds between atoms.

### 3.3 ADIOS *In Transit* Data Transport

ADIOS is a parallel I/O library with a POSIX-like API [13]. The API requests to write/read (or put/get) are separated from the engines that perform the requested services, allowing for a variety of optimizations for functionality and performance, such as *in transit* implementations utilizing RDMA interconnect hardware when available. SENSEI has implementations based on the 1.X and 2.X versions of the ADIOS API. The ADIOS 2.X series has a complete redesign of the internals in order to better prepare for exascale computing needs [14], so some terminology changes between these release versions. In ADIOS 1.X, *in transit* processing through memory-based staging methods is provided by several transports: FLEXPATH [6], Dataspaces [7], and DIMES [22]. Similarly, in ADIOS 2.X there are different engines that provide both *in transit* and *in situ* solutions: SST, SSC, and Inline.

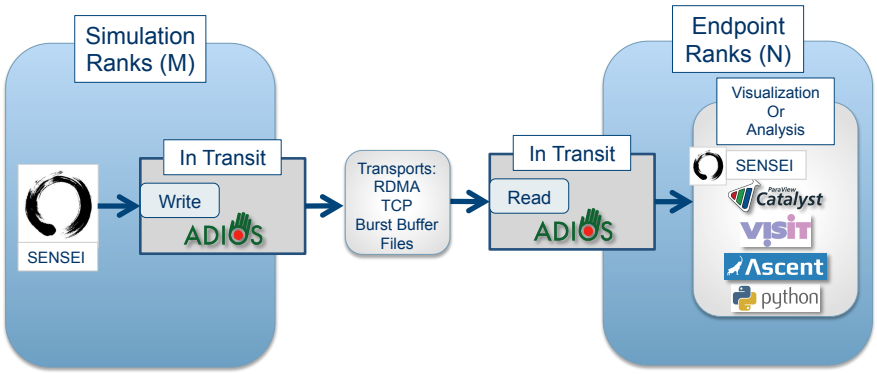



Fig. 8: The ADIOS transport layout in a SENSEI *in transit* use scenario. Image courtesy M. Wolf and W. Bethel.

From a SENSEI perspective, these differences only show up as slight differences in the configuration. A simple cartoon of this process and how ADIOS serves to connect them within SENSEI can be seen in Figure 8. ADIOS supports data movement and I/O through a number of engines[1]. When using ADIOS, SENSEI selects and configures the ADIOS engine based on settings in user provided XML.

```
<sensei>
  <transport type="adios1"
    filename="data_iso_has.bp"
     method="FLEXPATH" />
</sensei>
```

(a) The XML file used with both the simulation and the end-point to configure the ADIOS adaptor to use ADIOS 1.X API with the Flexpath transport.

```
<sensei>
  <transport type="adios2"
    filename="iso_has.sst"
    engine="sst" buffer_size="1">
  </transport>
</sensei>
```

(b) The XML file used with both the simulation and the end-point to configure the ADIOS adaptor to use ADIOS 2.X API with the SST transport.

Fig. 9: SENSEI XML used to select and configure ADIOS.

Figure 9 shows an example of utilizing the 2.X version of the ADIOS library with the SST transport for *in transit*. ADIOS provides both metadata queries and data selection methods. An ADIOS reader sees a global summary of all of the data available from all of the $M$ writers without respect to who originally wrote it. By default, when moving data between a simulation and an analysis job, SENSEI applies an equipartitioning algorithm mapping from the simulation's $P$ data blocks decomposed on the simulation's $M$ MPI ranks to the $N$ ranks of the analysis.

[1] called transports in ADIOS 1.x

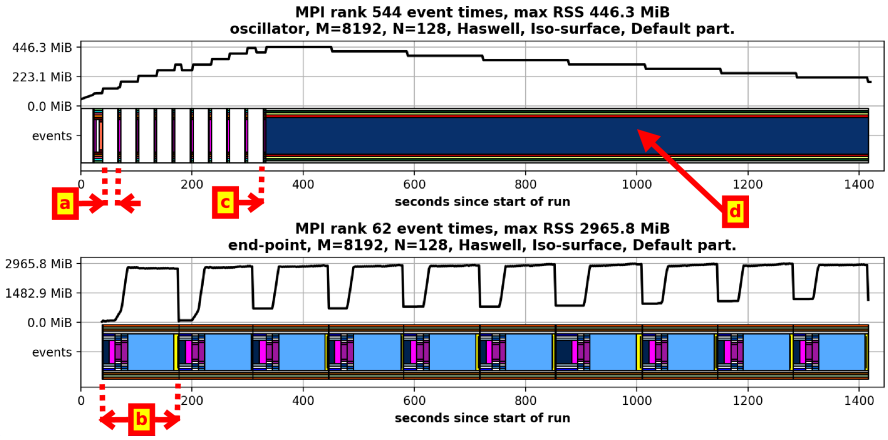



Fig. 10: ADIOS provided buffering hides the cost of a slow analysis from the simulation. Gantt charts from the simulation and analysis ranks with the largest resident set size (RSS) high water mark. Black line above the Gantt chart shows memory usage. Labels indicate: (a) time spent in the simulation computing one time step; (b) time spent in the analysis processing one time step; (c) simulation completes; and (d) ADIOS internally serves buffered data as fast as the analysis can consume it. Image courtesy B. Loring.

SENSEI's data model incorporates lightweight metadata that describe both the simulation data available and its mapping onto the simulation's MPI ranks. The availability of compact, cheap to move, metadata in *in transit* applications facilitates load balancing and re-meshing operations and makes it possible to improve the performance of *in transit* execution through judicious data downselection as described in Section 4.

Among a number of useful features, such as easily switching between conventional disk based I/O and network based data movement, ADIOS also includes advanced features such as dynamic disconnection and reconnection and a number of buffering controls. An example of the benefit of ADIOS buffering is shown in Fig. 10. Here, ADIOS hides the time spent by a slow analysis from the simulation by buffering data until the analysis can catch up. After each time step is buffered, control returns immediately to the simulation. The simulation completes in less than 400 seconds (label (c)) while the analysis continues well passed 1400 seconds. Were it not for ADIOS's buffering scheme, the simulation would run substantially slower.

# 4 Performance Analysis of SENSEI's M-to-N *In Transit* Infrastructure

We present summaries of two studies that evaluate the performance of the SENSEI M-to-N, *in transit* design and implementation. The main focus of the studies is to quantify the performance gains that can result when leveraging metadata so that only the data needed to solve a particular problem is moved from the producer to consumer. These studies were conducted at scale on an HPC system using a benchmark miniapplication from the SENSEI distribution (Section 4.1), and also using a state-of-the-art production science code that uses adaptive mesh refinement configured for a Rayleigh-Taylor instability calculation (Section 4.2). These studies reveal the types of significant performance gains that are made possible by having the data partitioning step being able to leverage metadata and consumer-side knowledge to make data requests from the consumer. The complete studies appear in other publications [15].

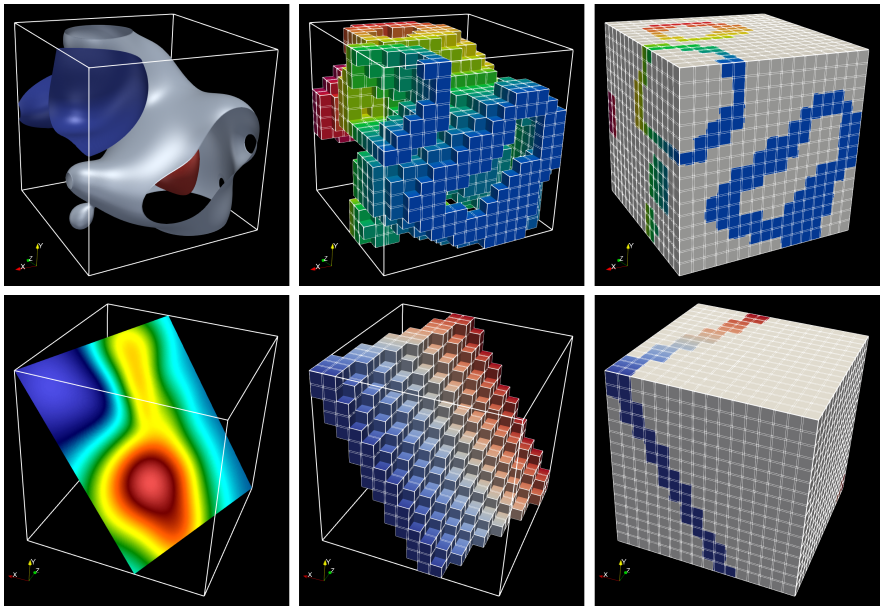
## 4.1 Data Source: Oscillators Miniapplication

One of the design objectives for our M-to-N architecture and implementation is to make it possible to move only the data that is needed for a particular operation. This section is a summarization of a study [15] that focuses on the cost savings that result when leveraging metadata to move only the relevant data needed to solve a problem.

The study focuses on two types of *in transit* operations where only a subset of data is required. One operation is a slice extract: in this case only those mesh cells or data blocks that intersect the slice plane are required to be moved to the *in transit* data consumer. The other operation is an isosurface extract: in this case, only those mesh cells that intersect the isosurface are required to be moved to the *in transit* data consumer. These operations are representative of two classes of methods that need only a subset of the original domain: one is based on some geometric constraints, in this case, a slice plane; the other is a data-dependent criteria, in this case, an isocontour.

The data producer is one of the SENSEI miniapplications, `oscillators`, which was run on NERSC's Cori Cray XC40 supercomputer. The code was configured to perform its computations on a $4096^3$ mesh, which was decomposed into 8192 blocks onto 8192 MPI ranks. Both `oscillators` and endpoints were configured to use the ADIOS 1.13.1 I/O library with the FLEXPATH staging method for data movement.

For each of the two extract operations, the study examines four different performance measures under varying levels of endpoint concurrency when using default and optimized partitioners. In this battery of tests, we wish to better understand a broad set of performance metrics and at varying levels of concurrency. The slice plane value and three isocontour levels are intended to result in non-trivial extracts, which are shown in the top portion of Figure 11. The left column shows the extracted geometry, the middle column shows the set of blocks that were used in the calcula-



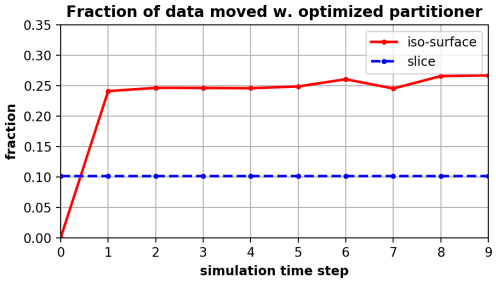



Fig. 11: Optimized partitioners determine which blocks are needed by the consumer and assign them to endpoint ranks while excluding all other blocks from the dataset. Top row: isosurface extract. Middle row: planar slice extract. Left column: extracted geometry. Center column: blocks needed to compute the extract. Right column: all blocks. Line plot: the fraction of simulation cells moved as a function of simulation time. This image is reprinted from our previous work [15].

tion, and the right columns shows all the blocks. When the optimized partitioner is in use, the white colored blocks were not moved from the simulation to the endpoint nor processed by the endpoint.

In the bottom portion of Figure 11, the chart shows, at varying simulation time steps, the fraction of total data moved from producer to consumer in each of the optimized partitioner configurations. For the three isosurfaces computed, only about 25% of the data contributes to a solution, and needs to be moved. An exception is at the first timestep, where the simulation has not evolved the computation to the point where the output has any cells that contain the isovalue. In the case of the slice, only about 10% of the mesh cells intersect the slice plane and contribute to the final solution.

The original study goes into much greater depth, examining multiple metrics at varying concurrency for the endpoints and using default and optimized partitioners. The study metrics include the amount of data moved from producer to consumer in bytes, the maximum memory footprint across all endpoint ranks, time to solution (which is akin to runtime), and cost of solution in terms of CPU hours.

Two of the lessons learned from that study are that different ratios of M-to-N produce vastly different results in terms of performance, and that the ratio of M-to-N varies depending on the problem. For example, with the isosurface endpoint, many more data blocks intersect the isocontour when compared to the slice extract, about 25% vs. 10%, as shown in Figure 11. As a result, when increasing the concurrency level of an endpoint, one encounters diminishing returns at different concurrencies, depending on the nature of the problem. There is no "one-size-fits-all" ratio that works best in all settings.

## 4.2 Data Source: AMReX-based IAMR Code

Continuing the summarization of an earlier study [15], we wish to examine the potential cost savings that might result when using an optimized rather than a default partitioner when using with a full-scale scientific simulation, as opposed to a miniapplication.

We instrumented the AMReX framework [23] for use with SENSEI. This gives us access to a wide variety of block structured adaptive mesh refinement (AMR) simulations for testing. In these experiments we made use of the AMReX-based IAMR compressible Navier-Stokes code [1] configured for the simulation of a Rayleigh-Taylor instability modeling the mixing of two fluids of different densities under the influence of gravity. The Rayleigh-Taylor instability produces a set complex isosurfaces that evolve in time.

In these runs, we configured IAMR for a base level of $1024^2$ x 2048 cells, one level of refinement, and ran on M=8192 ranks, with 4 OpenMP cores per rank, on 1025 KNL nodes of NERSC's Cori system. The total number of cores used by IAMR was 32768. The endpoint was run on 9 nodes with N=128 MPI ranks. We chose this ratio of $M$ to $N$ based upon the results of the miniapplication study in Section 4.1, which showed better performance gains for the optimized partitioner when $M >> N$.

One challenge in processing AMR data is that data blocks from refined levels duplicate and cover, either partially or fully, data blocks from coarser levels. Care must be taken when computing metadata and applying partitioning algorithms. For instance, in the calculation of per-block array minimum and maximum we use to determine if an isosurface intersects a block, we must not make use of data from the cells of that block that are covered by cells from a block in a more refined level. The reason is that covered cells are duplicated in the refined level and hence the isosurface will be duplicated as well. Our AMReX-specific data adaptor handles this aspect of the metadata calculation and as a result the optimized isosurface partitioner

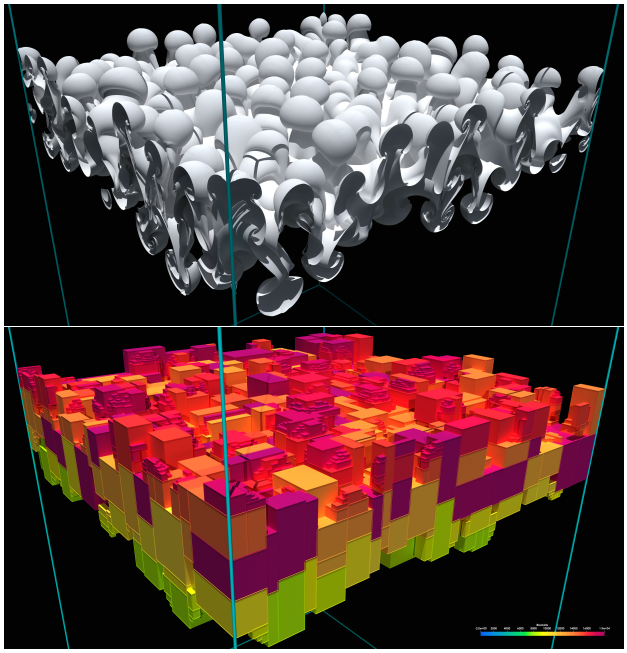


Fig. 12: Top: Isosurface extracted *in transit* from the AMR mesh produced by the IAMR code. Middle: Blocks, colored by block identifier, from the AMR mesh that contain the isosurface. These images are reprinted from our previous work [15].

can run without modification. The ability to handle complex dataset types such as AMR meshes, illustrates the flexibility of our approach.

The isosurface extracted at time step 420 is shown in Figure 12 in the top panel, along with the 4771 level-1 blocks that intersect this isosurface in the bottom panel. Figure 13 shows the amount of the data moved at each time step for isosurface extraction with the optimized partitioner (red line) compared to the total data size (blue line). In the worst case, the optimized partitioner moved less than 40% of the data.

The original study goes into more depth, including additional performance measures such as maximum memory footprint, time to solution, and cost of solution in terms of CPU hours [15]. Both the original study, and the summarization presented here, show consistency with those presented earlier in Section 4.1. Collectively, these studies show both the flexibility and performance gains that can result when leveraging rich metadata when solving a complex M-to-N data mapping problem.

## 5 Related Work

The idea of processing data as it is generated has been around for decades, with some of the earliest work consisting of a direct-to-film recording process from the 1960s. That work, along with a thorough survey of work in the *in situ* and *in transit* space is in a 2016 Eurographics STAR report [2].

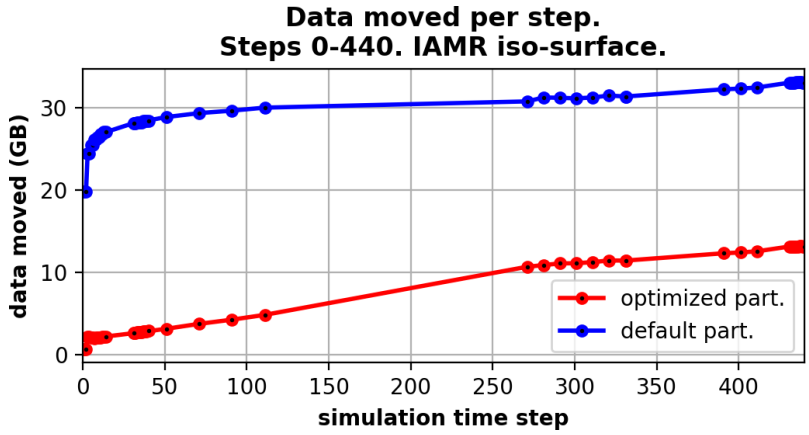



Fig. 13: Data moved during initial runs of the IAMR Rayleigh-Taylor problem. The data moved when the optimized partitioner is used (red line) is substantially less than the data that would be moved if a default partitioning algorithm is used (blue line). This image is reprinted from our previous work [15].

Early work on *in transit* infrastructure in the HPC space includes the CUMULVS project, which is middleware for coupling codes running at different levels of concurrency and for moving data between them in a M-to-N fashion [9]. A more recent CUMULVS report from 2006 [11] includes a survey of related projects focusing on M-to-N data partitioning and distribution on HPC platforms, some of which go back to the mid 1990s.

Over the years, several efforts have studied whether an *in situ* or *in transit* configuration will produce lowest cost, typically time-to-solution, for a given problem configuration. Oldfield et al., 2014 [18] evaluate *post hoc*, *in situ*, and *in transit* in the context of analysis and tracking of features in simulation output. They identify situations in which *in transit* or *in situ* approaches are more or less advantageous, such as *in transit* being advantageous when analysis computations are more complex and time consuming. Morozov and Lukić, 2016 [16] examine *in situ* and *in transit* configurations of a cosmological simulation coupled with a two-stage analysis pipeline, and find that when the analysis and simulation codes have different scaling properties, that it is advantageous to use *in transit* configurations. Kress et al., 2019 [12] study scalable rendering and aim to find the best balance of M to N producer and consumer ranks across different levels of concurrency by considering cost models for both *in situ* and *in transit* configurations.

The focus of our work here is on the design, implementation, and performance evaluation of a method for performing robust, flexible, and general purpose M-to-N data redistribution for use in an *in transit* setting at scale on HPC platforms. In particular, we are interested in understanding the performance gains that can result when leveraging metadata for partitioning and moving data from producer to consumer ranks in an *in transit* configuration. A similar idea appears in Childs

et al., 2005 [5], which describes the "contract" system in the VisIt application that results in optimizations of data movement through the visualization pipeline: downstream processing stages inform upstream stages of the data subsets needed to perform a specific computation. Our performance study uses some of the same use scenarios from that work, namely planar slicing and isocontouring, to illustrate the gains that result from optimizing data partitioning and movement. Whereas that earlier work measured and reported runtime improvement in the setting of an interactive GUI based post processing visualization application where the stages of their pipelines ran in the same process address space and data was provided by disk based I/O, our work here investigates similar approach applied in a setting where the data produced by a simulation is immediately moved to and processed in a separate application running at a different level of concurrency. In our work, fast interconnects move data and as such it never hits the disk: we look beyond just runtime to examine deeper levels of performance analysis. For each of the default and optimized partitioner configurations, we measure and report, in addition to runtime, the amount of data moved between producer and consumer ranks, and the memory footprint of producer and consumer ranks. These additional measurements beyond runtime provide significantly deeper insight into the benefit of the optimizations for *in transit* data partitioning and placement.

## 6 Conclusion and Future Work

In *in transit* processing, one of the central challenges is moving data from the $M$ producer ranks to the $N$ consumer ranks. We present a design pattern for a flexible, general purpose solution to this challenging problem. Our implementation adds new *in transit* capabilities to the SENSEI generic *in situ* interface, and we demonstrate its use and study its performance in a 32K-way parallel run of a production scientific simulation code, AMReX/IAMR, on a large HPC platform. Our performance evaluation measures runtime, amount of data moved, and time to solution to help reveal the nature of performance gains possible when using an optimized partitioner that moves only those portions of the data needed by the consumer.

A central theme of the design objectives is the idea of *proximity portability*, which means having the ability to run either *in situ* or *in transit*, and if run *in transit*, to be able to leverage any number of different potential tools for moving data between producer and consumer ranks. SENSEI's adaptor design pattern makes this possible, and this work shows use of three different mechanisms for moving data between *in transit* producers and consumers: HDF5, libiS, and ADIOS. This capability opens new avenues of research for furthering the advantages offered by an *in transit* approach, namely being able to overlap computation and communication, and being able to better load balance between $M$ simulation ranks and $N$ simulation ranks.

The combination of being able to repartition data from M-to-N producers and consumers and being able to perform runtime switching between different transports

is a critical building block that will accelerate further development and use of *in situ* and *in transit* methods for scientific computing. We anticipate an explosive growth in applications of methods for doing learning, analysis, and cooperative computing when science code teams are easily able to couple codes in a flexible way as shown by the design principles, examples, and performance studies we have presented in this chapter.

## References


1. Almgren, A.S., Bell, J.B., Colella, P., Howell, L.H., Welcome, M.L.: A conservative adaptive projection method for the variable density incompressible navier–stokes equations. Journal of Computational Physics **142**(1), 1 – 46 (1998). DOI https://doi.org/10.1006/jcph.1998.5890. URL http://www.sciencedirect.com/science/article/pii/S0021999198958909
2. Bauer, A.C., Abbasi, H., Ahrens, J., Childs, H., Geveci, B., Klasky, S., Moreland, K., O'Leary, P., Vishwanath, V., Whitlock, B., Bethel, E.W.: *In Situ* Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report. Computer Graphics Forum (Special Issue: Proceedings of EuroVis 2016) **35**(3) (2016). LBNL-1005709
3. Camp, D., Garth, C., Childs, H., Pugmire, D., Joy, K.I.: Streamline integration using mpi-hybrid parallelism on a large multicore architecture. IEEE Transactions on Visualization and Computer Graphics **17**(11), 1702–1713 (2011). DOI http://doi.ieeecomputersociety.org/10.1109/TVCG.2010.259
4. Cherukara, M.J., Narayanan, B., Chan, H., Sankaranarayanan, S.K.R.S.: Silicene growth through island migration and coalescence. Nanoscale **9**, 10186–10192 (2017). DOI 10.1039/C7NR03153J. URL http://dx.doi.org/10.1039/C7NR03153J
5. Childs, H., Brugger, E., Bonnell, K.S., Meredith, J.S., Miller, M.C., Whitlock, B., Max, N.L.: A Contract-Based System for Large Data Visualization. In: Proceedings of IEEE Visualization (Vis05). Minneapolis, MN (2005)
6. Dayal, J., et al.: Flexpath: Type-based publish/subscribe system for large-scale science analytics. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 246–255. IEEE (2014)
7. Docan, C., Parashar, M., Klasky, S.: Dataspaces: an interaction and coordination framework for coupled simulation workflows. Cluster Computing **15**(2), 163–181 (2012)
8. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An overview of the HDF5 technology suite and its applications. In: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases, pp. 36–47. ACM (2011). Software at http://www.hdfgroup.org/HDF5/
9. Geist, G.A., Kohl, J.A., Papadopoulos, P.M.: CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. International Journal of High Performance Computing Applications **11**(3), 224–236 (1997)
10. Gu, J., Loring, B., Wu, K., Bethel, E.W.: HDF5 as a Vehicle for in Transit Data Movement. In: Proceedings of the SC19 Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, ISAV '19, p. 39–43. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3364228.3364237. URL https://doi.org/10.1145/3364228.3364237
11. Kohl, J.A., Wilde, T., Bernholdt, D.E.: Cumulvs: Interacting with high-performance scientific simulations, for visualization, steering and fault tolerance. The International Journal of High Performance Computing Applications **20**(2), 255–285 (2006)
12. Kress, J., et al.: Comparing the efficiency of in situ visualization paradigms at scale. In: International Conference on High Performance Computing, pp. 99–117. Springer (2019)
13. Liu, Q., et al.: Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. Concurrency and Computation: Practice and Experience **26**(7), 1453–1473 (2014)

14. Logan, J., Ainsworth, M., Atkins, C., Chen, J., Choi, J.Y., Gu, J., Kress, J.M., Eisenhauer, G., Geveci, B., Godoy, W., et al.: Extending the publish/subscribe abstraction for high-performance i/o and data management at extreme scale. Bulletin of the Technical Committee on Data Engineering **43**(1) (2020)
15. Loring, B., Gu, J., Ferrier, N., Rizzi, S., Shudler, S., Kress, J., Logan, J., Wolf, M., Bethel, E.W.: Improving performance of m-to-n processing and data redistribution in in transit analysis and visualization. In: EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV). Norrköping, Sweden (2020)
16. Morozov, D., Lukić, Z.: Master of puppets: Cooperative multitasking for in situ processing. In: Proceedings of the Symposium on High-Performance Parallel and Distributed Computing (HPDC), pp. 285–288 (2016)
17. Mortensen, M., Dalcin, L., Keyes, D.: mpi4py-fft: Parallel fast fourier transforms with mpi for python. Journal of Open Source Software **4**, 1340 (2019)
18. Oldfield, R.A., et al.: Evaluation of methods to integrate analysis into a large-scale shock shock physics code. In: Proceedings of the 28th ACM International Conference on Supercomputing, ICS '14, pp. 83–92 (2014)
19. Usher, W., Rizzi, S., Wald, I., Amstutz, J., Insley, J., Vishwanath, V., Ferrier, N., Papka, M.E., Pascucci, V.: 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, p. 33–38. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3281464.3281466. URL https://doi.org/10.1145/3281464.3281466
20. Usher, W., Rizzi, S., Wald, I., Amstutz, J., Insley, J., Vishwanath, V., Ferrier, N., Papka, M.E., Pascucci, V.: libis: A lightweight library for flexible in transit visualization. In: 2018 Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV) (2018)
21. Wald, I., Johnson, G.P., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J., Günther, J., Navrátil, P.: Ospray-a cpu ray tracing framework for scientific visualization. IEEE transactions on visualization and computer graphics **23**(1), 931–940 (2016)
22. Zhang, F., et al.: In-memory staging and data-centric task placement for coupled scientific simulation workflows. Concurrency and Computation: Practice and Experience **29**(12), e4147 (2017)
23. Zhang, W., Almgren, A., Beckner, V., Bell, J., Blaschke, J., Chan, C., Day, M., Friesen, B., Gott, K., Graves, D., Katz, M.P., Myers, A., Nguyen, T., Nonaka, A., Rosso, M., Williams, S., Zingale, M.: Amrex: a framework for block-structured adaptive mesh refinement. Journal of Open Source Software **4**(37), 1370 (2019). DOI 10.21105/joss.01370. URL https://doi.org/10.21105/joss.01370