A Data Streaming Model in MPI

Ivy Bo Peng, Stefano Markidis, Erwin Laure
KTH Royal Institute of Technology
Stockholm, Sweden
bopeng@kth.se
markidis@kth.se
erwinl@kth.se

Daniel Holmes, Mark Bull
EPCC, University of Edinburgh
Edinburgh, UK
dholmes@staffmail.ed.ac.uk
markb@epcc.ed.ac.uk

ABSTRACT

Data streaming model is an effective way to tackle the challenge of data-intensive applications. As traditional HPC applications generate large volume of data and more data-intensive applications move to HPC infrastructures, it is necessary to investigate the feasibility of combining message-passing and streaming programming models. MPI, the de facto standard for programming on HPC, cannot intuitively express the communication pattern and the functional operations required in streaming models. In this work, we designed and implemented a data streaming library *MPIStream* atop MPI to allocate data producers and consumers, to stream data continuously or irregularly and to process data at run-time. In the same spirit as the STREAM benchmark, we developed a parallel stream benchmark to measure data processing rate. The performance of the library largely depends on the size of the stream element, the number of data producers and consumers and the computational intensity of processing one stream element. With 2,048 data producers and 2,048 data consumers in the parallel benchmark, *MPIStream* achieved 200 GB/s processing rate on a Blue Gene/Q supercomputer. We illustrate that a streaming library for HPC applications can effectively enable irregular parallel I/O, application monitoring and threshold collective operations.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Design, Performance

Keywords

MPI, streaming model, data-intensive, HPC

ExaMPI2015, November 15-20, 2015, Austin, TX, USA
@2015ACM. ISBN 978-1-4503-3998-8/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2831129.2831131.

1. INTRODUCTION

Traditional High Performance Computing (HPC) deals with compute-intensive applications. HPC applications are expected to run on billion processes and generate enormous volume of data in the coming exascale era. One example is the VPIC code running on Blue Waters generated 290TB data on 300,000 MPI processes [2]. Lately, data analysis and analytics of large data sets are an emerging class of applications in the HPC realm as the exponential growth of data size demands for HPC infrastructures to process within reasonable time. For example, the Square Kilometer Array will provide approximately 200 GBytes/s raw data [21]. For this reason, the software frameworks for data analytics on clusters are gaining popularity. HPC hardwares, parallel programming models and systems are evolving to tackle the challenges from data-intensive applications. It has been said that HPC and Big Data are *converging* [17].

One of the open issues in the convergence of HPC and Big Data is the lack of a main HPC programming system that can fully and conveniently express the irregular and fine-grained communication in data-intensive applications. MPI is the de-facto standard for programming parallel applications running on supercomputers. It is important to discuss whether MPI requires any modifications or additions to effectively support the shift from compute-intensive to data-intensive applications. In fact, irregular and fine-grained communication is not perfectly suited to the message-passing programming model. In this paper, we discuss the possibility to provide new MPI features to support the data streaming model.

The data streaming model has origins in the 80's to address the problem of processing data on computers with limited memory [11, 12]. A data stream transports elements from a source (a data structure, an array or an I/O channel) through a pipeline of operations. For this reason, a data stream is not a data structure nor an array and it does not require storage. Stream has a *functional* nature as operations defined on the stream do not modify its source when producing a result. The goal of the stream model is to compute a function from the stream, e.g. to analyze the distribution function from a stream of log data. The data streaming model requires a single-pass analysis: stream elements are *consumable* so that each element is only accessed and processed once. On the other hand, MPI is based on the concept of messages: data is encapsulated in message envelopes containing information of source, destination, tag and communicator. The receiver process of a message is responsible for allocating memory space to hold the incoming data. For

this reason, receiving irregular data of unknown size from unknown senders, though common in the data streaming model, is still a challenge in MPI.
The goal of this work to investigate the possibility of combining the message-passing and data streaming model. We present our approach of supporting functional-style operations, such as map-reduce and filtering operations, on stream elements in MPI. For a proof-of-concept, we designed and implemented a data streaming library atop MPI. We benchmark the performance of our library varying the number of MPI processes, the size of stream element and the granularity of the operation on the stream element. The benchmarking results show acceptable performance (52%-65% of the maximum available bandwidth) reaching 200 GB/s processing rate on BG/Q supercomputer. We discuss in details some HPC applications that can benefit from using a data streaming model in MPI.
The paper is organized as follows. Section 2 presents previous work on programming approaches for data streaming models in MPI. Section 3 describes the design and the implementation of a library to support data streaming in MPI. Performance evaluation of our library is also present in Section 4. Section 5 describes possible use cases of the MPI data streaming model in HPC applications. Section 6 concludes the paper and outlines future work.

2. RELATED WORK

Spark Streaming [22] is a popular cluster computing framework that supports iterative machine learning algorithms and interactive data analysis. StreamIt [18] language enables high-performance streaming applications. Spark is implemented in Scala while StreamIt relies on the StreamIt compiler for high performance. However, most HPC scientific applications based on MPI are programmed in C or Fortran and optimised for high-performance computation. The goal of our current work is to support streaming model in these MPI based HPC applications with minimal reconstruction to the application. Ref. [7] presented a hybrid MPI-streaming model to combine MPI for intensive local computation with the streaming model for heterogenous communication systems (mixing intra and inter cluster communication). This hybrid model uses MPI to link independent kernels by using the output of one kernel as the input of another one. Different from their work, our work is to develop a streaming model within MPI and focuses on the functional operations on the stream elements. Our streaming model has semantic differences such that a stream consists of multiple stream elements from the data producers to consumers and only one operation is defined per stream.
Ref. [5] proposed an optimized implementation of the map-reduce model on top of MPI, using non-blocking reduction and local-reduction operations. The application of MPI in studying large graph problems has been presented in [16]. DataMPI [6] proposed an extension of MPI for processing and communicating a large number of key-value problems. It implements a bipartite key-value pair based communication library in the same spirit as the Hadoop-like computing. In DataMPI, data are moved from one set of tasks (maps) to the other set (reduces). These two groups of tasks and the data movement between them form a bipartite graph with dichotomic processes (the two groups of tasks belong to different MPI communicators). While DataMPI focuses on MPI-enabled work-sharing, our model targets for a loosely coupled relationship between the data producers and consumers and a stream-enabled global view of all processes.
Ref. [19] proposed Active Messages (AM), an asynchronous communication mechanism that embeds a handle in the message head and triggers the handle upon the arrival of messages. This model exploits the privileged interrupt handler on message-driven machines and is suitable for data-intensive application. An MPI-interoperable framework was proposed by [23]. An AM framework targets for operations local to the data in one message while our model targets for operations on one data stream consisting of multiple stream elements from a group of data producers.

3. AN MPI STREAMING LIBRARY

3.1 Design

In this work, we focus on data stream models for distributed systems where MPI is the main programming system. For this reason, we mainly consider *parallel* streams: there are one or more stream sources distributed among different processes, called *data producers*. Data relative to one parallel stream is divided among different data producers. A *data consumer* is a process that receives and processes the elements of a parallel stream. Communication between the data producers and consumers is needed for transmitting the stream elements. In our model, a stream *channel* consists of a group of data producers and consumers sharing the same communicator and exchanging information via persistent communication.
Data consumers process the incoming elements of a parallel stream on a first-come-first-served basis following defined operations. One or more operations can be *attached* to a parallel stream. Two kinds of operations are supported: *intermediate* and *terminal*. Intermediate operations produce a result local to one data consumer, i.e. the result depends only on the stream elements received by that process. A terminal operation is a collective reduction operation called by all data consumers and produces a final global of the parallel stream.
Our data streaming model provides the flexibility to support both *stateless* and *stateful* operations on stream elements. Stateless operations do not rely on any information from previously processed elements, e.g. filtering based on a threshold. Stateful operations usually require higher memory usage than stateless operations as they need to store information from previously processed stream elements, e.g. sorting stream elements. One naive solution could be using the intermediate operation to store all received elements in the result buffer and using the terminal operation to carry out a merge sort on the stored elements on all data consumers.
Our data streaming model does not guarantee the *encounter order* of the data streams. Stream elements will not be processed in the order they are streamed out but in the order they arrive on a data consumer. After a stream has been terminated, it is not supported to restart the same stream.
Figure 1 outlines our data streaming model. Eight data producers (light blue squares denoted with P) transmit one parallel stream to two data consumers (green squares denoted with C). The parallel stream consists of multiple stream elements (yellow rectangles) contributed by different data producers (indicated by the dashed arrows). Each data producer could stream out different number of stream elements

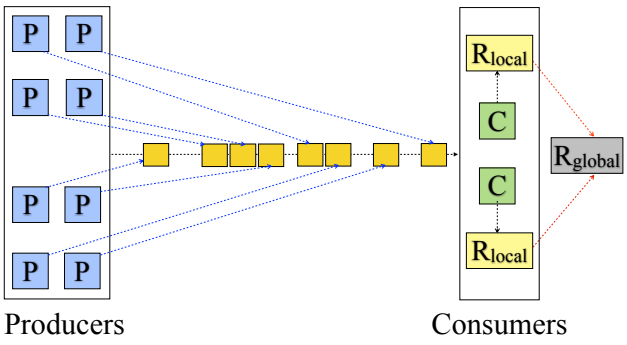

Figure 1: Diagram of a streaming model with eight data producers contributing to one parallel stream and two data consumers. Each data consumer keeps on processing their share of stream elements and updating a local result (R_{local}). When the stream terminates, a global result (R_{global}) is produced based on the local results on all data consumers.

at their own pace. The two data consumers share the workload of processing the parallel stream. Each data consumer processes the elements it receives and updates to its local result. At the termination of a stream, a terminal operation combines the local results on all data consumers to a final global result.

3.2 Implementation

Our implementation of an MPI streaming library, called *MPIStream*, was written in C and built on the top of MPI. The MPIStream library maintains information about the data streaming environment in two objects:

1. `MPIStream_Channel` object. This object stores the information about the group of data producers and consumers. It identifies whether an MPI process is a data producer or a consumer. It maintains two communicators `StreamOpComm` and `StreamGlobalComm`. Information on the allocated data consumers is stored on data producers while the size of the allocated data producers is maintained on data consumers. This object also maintains the list of parallel streams attached to this channel.

2. `MPIStream` object. This object stores the information of one parallel stream. The main parts include the data structure of the stream elements and the result as well as the operations for processing each stream element. On the data consumers, this object also maintains a memory buffer and an array of MPI persistent requests for receiving multiple incoming stream elements at a time.

Multiple parallel streams can be attached to one stream channel. In this way, different data can be streamed out and processed using different operations on the same group of data producers and consumers.

The MPIStream library maintains two MPI communicators to isolate the communication for streaming data from the communication for the application. The `StreamOpComm` communicator only consists of data consumers while the `StreamGlobalComm` communicator maintains the union of data producers and consumers. These two communicators are created with `MPI_Comm_Split()`. In our implementation, the association between the data producers and consumers is static within one stream channel and transparent to the user. The `StreamGlobalComm` communicator enables intra-communicator communication between the data producers and consumers. The `StreamOpComm` communicator has a smaller size than `StreamGlobalComm` so that the collective reduction operations among all data consumers can be speeded up.

In our MPI streaming model, the stream elements are continuously received and processed by the data consumers. To reduce the overhead of repeatedly calling the point-to-point receive routine, we use persistent communication on the data consumers. Persistent communication binds a list of arguments to a persistent communication request and this request can be reused to start and complete a message [4].We note that the concept of *persistent communication* as a repeated communication with the same argument list is the closest concept to the *stream* communication in MPI.

Multiple data producers can contribute to one parallel data stream. This requires a data consumer to be able to handle multiple incoming stream elements at a time. In our implementation, each data consumer maintains a memory buffer large enough to hold multiple stream elements and a list of persistent communication requests. Figure 2 illustrates a data consumer that maintains a memory buffer capable of holding five stream elements and a list of five persistent requests, each of which has its receive buffer pointing to the respective location in the memory buffer. The data consumers then use the `MPI_Testany()` function to detect the completion of any requests in the list and process the incoming stream elements on the first-come-first-served basis. We use `MPI_Testany()` instead of `MPI_Testsome()` or `MPI_Waitsome()` as there is a separate queue to receive query from data producers with higher priority.

Our library allows the application to decide the ratio between the data producers and consumers. To achieve scalability when this ratio increases to a large number, no specific information about the data producers is maintained on the data consumers. Our implementation uses the wildcard `MPI_ANY_SOURCE` in the persistent requests to receive stream elements from any data producers. However, this introduces clashes on message matching as one incoming message can match every persistent request in the list. We use unique message tags for each persistent request to overcome the clash issue. Figure 2 demonstrates that data consumer 0 maintains five persistent requests, $Rq0$ to $Rq4$. Each request has a unique tag in the range 1 to 5. When data consumers 0, 1 and 2 stream out elements, a random tag within the range 1 to 5 is associated to each stream element. This tagging operation is done transparently to the user. There are two advantages of using randomized tags. First, it is a local operation without communication between data producers and consumers. Second, it spreads incoming messages evenly on the persistent requests list to reduce potential contention raised from multiple messages matching the same receive request.

Figure 2 shows that stream element 1, 2, 3 are streamed out approximately at the same time but will match different receive requests that point to different locations in the memory buffer. $Rq0$ has two matching stream elements, stream element 0 and 1. As stream element 0 arrives earlier, it would likely have been processed before the arrival of stream element 1.

Stream elements are the basic building blocks of a paral-

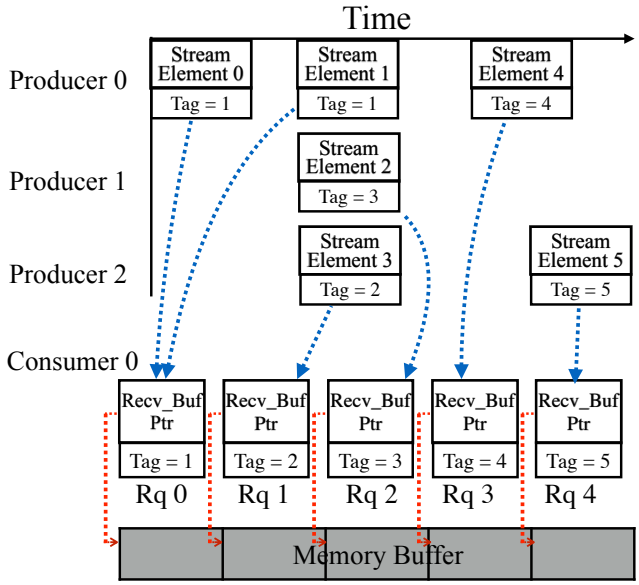

Figure 2: Diagram for memory buffer and persistent requests management in each stream operation. Each data consumer maintains a list of persistent requests and a memory buffer capable of holding multiple stream elements. Unique tags are used in the persistent requests. Data producers generate a random tag when streaming out an element.

lel stream and define the unit of transmission between the data producers and consumers. Our library uses MPI data types to specify the data layout of stream elements on data producers. MPI data type allows us to achieve zero-copy streaming even when the memory layout of the stream element on the data consumer is non-contiguous. For instance, a derived MPI data type can be defined to describe the outer layer of a sub-mesh. When this data type is used to describe the stream element, a data producer does not need to allocate a separate buffer for streaming data. MPI data types are also used to describe the data layout of results on data consumers. For example, the result could be an array of integers or a mixture of different data types. When stream elements and results are collections of the same primitive data types, it could be useful to define a process data type. For example, if the stream element consists of four integers and the result consists of two integers, defining a process data type of integer enables array operation on the stream elements.

The operation used for processing each stream element consists of two parts: an intermediate and a terminal function. The operations are defined in a `MPIStream_Operation` structure. For both functions, it could be either an MPI predefined function or a user-defined callback function. The intermediate function is called when a stream element arrives and only updates the local result. The terminal function is called by all data consumers in a collective reduction operation at the termination of a parallel stream. For this reason, the user-defined function in the terminal function needs to follow the prototype for the user-defined operations in MPI reduction operation.

In a streaming model, it is common that the actual size of a stream is only known at run-time. The MPIStream library does not require prior information about the stream length and supports run-time termination of parallel streams. Each data consumer continues streaming out data until it signals the termination of its contribution to a parallel stream. When a data consumer receives termination signals from all data producers associated to it, it stops receiving incoming stream elements and calls the terminal function.

The basic functions supported in the MPIStream library are:

1. `int MPIStream_CreateChannel(int isProducer, int isConsumer, MPI_Comm comm, MPIStream_Channel* channel)`. This is a collective operation. Every MPI process in `MPI_Comm` comm needs to call. The `MPIStream_Channel` objects is set up on processes that are either data producer or consumer.

2. `int MPIStream_FreeChannel(MPIStream_Channel* channel)`. This operation frees the communicators set up for the stream communication and deallocates memory.

3. `int MPIStream_Attach(MPI_Datatype streamDataType, MPI_Datatype resultDataType, MPI_Datatype processDataType, MPIStream_Operation *operation, MPI_Stream *stream, MPIStream_Channel *channel)`. This function attaches a parallel stream to a stream channel. This function specifies a parallel stream, i.e. the data type of its stream elements, the operations for processing its stream elements and the channel it attaches to. It will setup an `MPIStream` object so that the user can retrieve information about the parallel stream.

4. `int MPIStream_Send/MPIStream_Isend(void* sendbuf, MPI_Stream *stream)`. These functions allow a data producer to stream out one stream element belonging to the specified parallel stream.

5. `int MPIStream_Terminate/MPIStream_Iterminate(MPI_Stream stream)`. This function is called by a data producer to signal the termination of its contribution to the specified parallel stream.

6. `int MPIStream_Query/MPIStream_Iquery(void* resultbuf, int *flag, MPI_Stream *stream)`. In certain cases, the data producer might need to retrieve an intermediate result of the previously streamed elements. This function is called by a data producer to query the data consumers about the current global result. Note that this function will trigger a collective reduction operation among data consumers and is not recommended for frequent use.

7. `int MPIStream_ThresholdColl(void *local_result, void *global_result, MPI_Stream *stream)`. This is a collective operation that supports early termination of a reduction operation without requiring all data producers to participate if a conclusion can be drawn, e.g. a specified threshold value is exceeded.

8. `int MPIStream_Operate/MPIStream_Ioperate(MPI_Stream* stream)` These functions are called by a data consumer to receive and process the incoming stream elements.

Listing 1 shows a simple example of using the MPIStream library to accumulate stream elements of one integer from all data producers. During initialization, all the processes create a channel and attach a parallel stream to that channel. During iteration, data producers keep on streaming out

Listing 1: Example application using the MPIStream library

```
#include "MPIStream.h"
int main(int argc, char** argv)
{
    int myrank, nprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    // two data consumers, the rest are data producers
    int is_data_producer = myrank < (nprocs-2);
    int is_data_consumer = myrank >= (nprocs-2);
    // stream initialization stage
    MPIStream_Channel channel;
    MPIStream_CreateChannel(is_data_producer,is_data_consumer,MPI_COMM_WORLD, &channel);
    //  1. specify stream element
    MPI_Datatype resultDataType,streamDatatype,processDatatype;
    processDatatype = MPI_INT;
    streamDatatype = MPI_INT;
    resultDataType = MPI_INT;
    // 2. specify one stream operation
    MPIStream_Operation operation;
    operation.intermediate_function = &AccumulateOverTime;
    operation.terminal_function          = MPI_SUM;
    // 3. attach the specified operation to a parallel stream
    MPIStream stream0;
    MPIStream_Attach(streamDatatype,resultDataType,processDatatype,&operation,
          &stream0,&channel);
    // Start of the data streaming stage
    if(is_data_consumer) // data consumers keep on processing data
        MPIStream_Operate(&stream0);
    if(is_data_producer){
        for(int i = 0; i<100;i++) // data producers stream out 100 elements belonging to
              stream0
            MPIStream_Send(&i,&stream0);
        MPIStream_Terminate(&stream0); // data producers signal the termination of its data
    }
    // Finalize
    MPIStream_FreeChannel(&channel);
    MPI_Finalize();
    return 0;
}
// user-defined operation on stream
void AccumulateOverTime(void *in, void *inout, int *len, MPI_Datatype *datatype)
{
    if(*len == 0)
        *(int*)inout = 0; //initialize the result
    else if(*len == 1){
        *((int*)inout) += *((int*)in); //accumulate the result
    }
 }
```

some data. When a data producer finishes sending all its data, it signals the termination. The data consumers keep on receiving and processing the incoming stream data until all its allocated data producers have signalled termination. Figure 3 presents the tracing of the MPIStream application shown in Listing 1. The tracing is obtained by using the MPE tool [3]. The initialization stage is indicated in the green and yellow blocks at the beginning on each process. After that, data producers start their local work (indicated in the white spaces) and stream out data at different rates (indicated in the dark blue bars). The two data consumers keep on polling for the arrival of stream elements (indicated in the black bars). If so, the data consumers process the data (indicated in the white spaces) and restart the persistent request to receive new elements (indicated in the purple bars).

The interoperability of a data streaming model with an application is straightforward and adaptable. Each MPI process can take up the role as a data consumer or data consumer and only function when needed. In this way, individual application can decide the optimal ratio between the number of data producers and consumers depending on the nature of the streaming data, e.g. the frequency of streaming vs the operation time on each stream element. For example, if the defined operation on each stream element takes longer than the average time interval between two incoming stream elements, a higher ratio of number of data consumers to number of data producers is desirable.

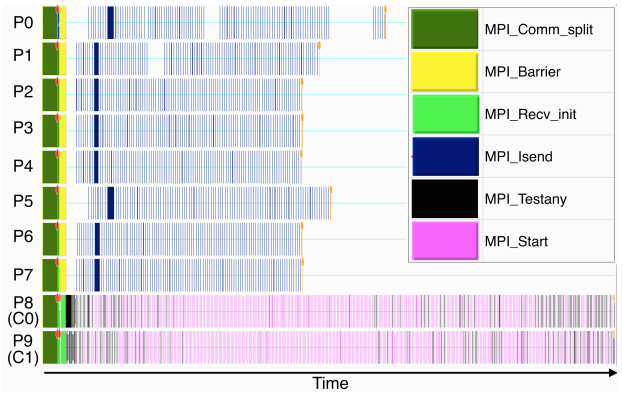

Figure 3: MPE tracing of the application presented in Listing 1 on 10 MPI processes. The first eight processes are data producers and stream out the data at difference rates. The last two processes are data consumers and keep on receiving and processing data.

4. PERFORMANCE EVALUATION

We evaluated the performance of the MPIStream library with a test-suite similar in nature to the popular STREAM benchmark [10]. The original STREAM benchmark is used to measure the sustainable memory bandwidth. We extended it to a parallel STREAM benchmark to measure the *processing rate* of data consumers in an application using the MPIStream library. In the same spirit as the STREAM benchmark, we implemented four computational kernels: *copy*, *scale*, *add* and *triad*. A stream element of different sizes and in different configurations of data producers per consumer is streamed 10,000 times. We use the *scale* kernel and `MPI_DOUBLE` as the elementary stream type. Similar results can be obtained using the other three kernels. For each test, we ensure the communication between the data producer and data consumer processes is inter-node communication and report the maximum and minimum processing rate over a series of 20 runs.

The parallel STREAM benchmark shows the amount of data processed per second and also measures the interconnection network bandwidth utilized by the MPIStream library. We compare the bandwidth utilized by the MPIStream library with the available bandwidth calculated with a simple *ping-pong* benchmark. We noticed that the bandwidth utilization by the MPIStream library is a fraction of the available bandwidth as the the receiving (using MPI wildcard and polling operations) and processing stream elements at data consumer side comes with an overhead.

4.1 Benchmark Environment

We carried out the parallel STREAM test-suite on two different computational testbeds. First, we used the KTH Beskow supercomputer. Beskow is a Cray XC40 system, based on Intel Xeon E5-2698v3 16-core (2.30 GHz) processors and Cray Aries interconnect network with Dragonfly topology. Each Beskow node has 32 cores divided between two sockets, with 16 cores on each. The RAM for each node is 64 GB. The total number of cores is 53,632. Cray C compiler version 5.2.40 and the Cray MPICH2 library version

7.0.4 have been used. Second, we used the Argonne National Laboratory Vesta supercomputer. Vesta is an IBM Blue Gene/Q supercomputer with 2,048 nodes with 16 core 1.6 GHz PowerPC CPU and 16 GB RAM per node. Vesta uses a 5D Torus interconnect network. The IBM XL C/C++ compiler for Blue Gene version 12.1 and IBM MPI2.2, based on MPICH, were used for the test.

4.2 Benchmark Results

In the first test, we analyze the dependence of the processing rate on different configurations of the number of data producers per consumer. We have a fixed number of 32 data consumers running on 1 Beskow's node and 2 Vesta's nodes. We increase the number of data producers per consumer from 1 to 25, i.e. a total of 800 data producers and 32 data consumers. In addition, we studied the impact of varying the size of the stream element from 256 doubles (2,048 bytes) to 1,024 doubles (8192 bytes).
The top panel in Figure 4 shows the maximum and minimum processing rate in GB/s on the Cray XC40 supercomputer. For the stream element size of 256 doubles and 512 doubles, the processing rate is almost constant at 9 GB/s increasing the number of data producers per consumer to 16. This corresponds to the use of 512 data producers and 32 data consumers. The processing rate for the stream element size of 256 doubles and 512 doubles drops by almost ten times when more than 16 data producers per consumer process are in use. If the stream element has a size of 1,024 doubles, the processing rate decreases almost linearly from 5GB/s (one data producer per consumer) to 3GB/s (25 data producers per consumer). For the stream element size of 1,024 doubles, we did not observe any sharp decrease in the processing rate in the test.
The bottom panel in Figure 4 shows the maximum and minimum processing rate in GB/s on the Vesta Blue Gene/Q supercomputer. For the stream element size of 256 doubles and 512 doubles, the processing rate drops exponentially from 1.5 GB/s and 3.2 GB/s to approximately 0.8 GB/s and 1.5 GB/s respectively when increasing the number of data producers per consumer from one to five. In the case of the stream element size of 1,024 doubles, the processing rate increases from 3.2 GB/s to 5 GB/s when increasing the number of data producers per consumer from one to five. For all the three stream element size, the processing rate remains almost constant if there are more than five data producers per consumer.

In the second test, we studied the impact of the computational intensity of processing each stream element on the processing rate. We varied the number of FLoating-point OPerations (FLOPs) per stream element. We carried out the test on the Cray XC40 and the Blue Gene/Q supercomputers with 32 data producers over 32 data consumers, using three stream element sizes: 256 doubles, 512 doubles and 1,024 doubles. We observe approximately constant processing rate on both machines when increasing FLOPs per stream element to a certain number (10,000 on Cray XC40 and 100 on Blue Gene/Q). Beyond that number of FLOPs, the processing rate drops almost linearly. This could be due to the fact that in the case of very large FLOPs per stream element, computation time on each element stream at the consumer side can only be partially overlapped with communication, decreasing the processing rate.

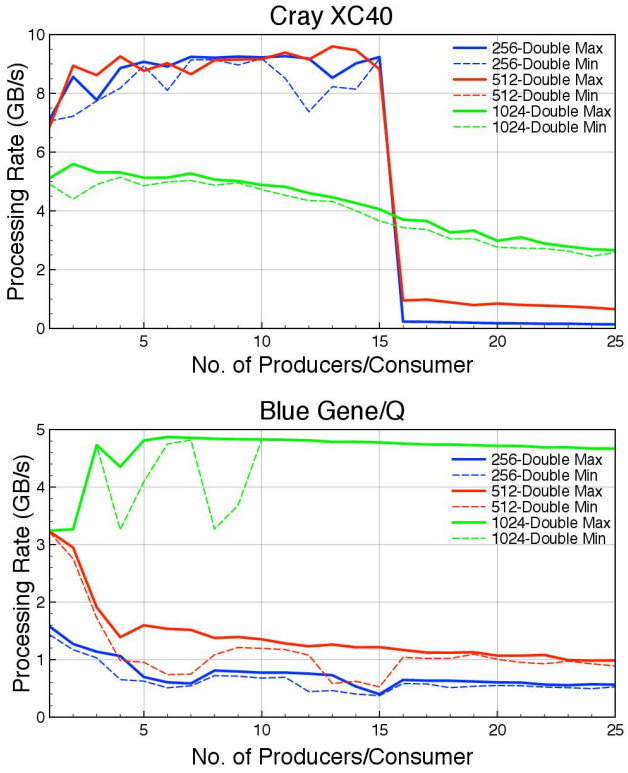

Figure 4: The maximum and minimum processing rate in GB/s varying the number of data producers per consumer using the *scale* benchmark. The number of data consumers is 32. The top panel presents results on a Cray XC40 supercomputer. The bottom panel presents results on a Blue Gene/Q supercomputer.

The top panel in Figure 5 shows the maximum and minimum processing rate in GB/s varying the FLOPs per stream element using the *scale* benchmark on the Cray XC40 supercomputer. We observe that the maximum processing rate is 8GB/s for the stream element size of 512 doubles. The processing rate for different stream element sizes is nearly constant (approximately 7GB/s) up to 10,000 FLOPs per stream element and then it drops.
The bottom panel in Figure 5 shows the maximum and minimum processing rate in GB/s varying the FLOPs per stream element using the *scale* benchmark on the Vesta Blue Gene/Q supercomputer. We note that the processing rate keeps approximately constant at 3.25GB/s for different element size until 100 (stream element size of 256 doubles and 512 doubles) and 1,000 (stream element size of 1,024 doubles) FLOPs per stream element.

In the third test, we studied the impact of the stream element size on the processing rate and the bandwidth utilization by the MPIStream library. We varied the stream element size from one double to 4,194,304 doubles. We carried out the test on the Cray XC40 and the Blue Gene/Q supercomputers with 32 data producers over 32 data consumers using the *scale* benchmark. We calculate the available bandwidth using *ping-pong* test under the same system configuration of the parallel STREAM benchmark. We ob-

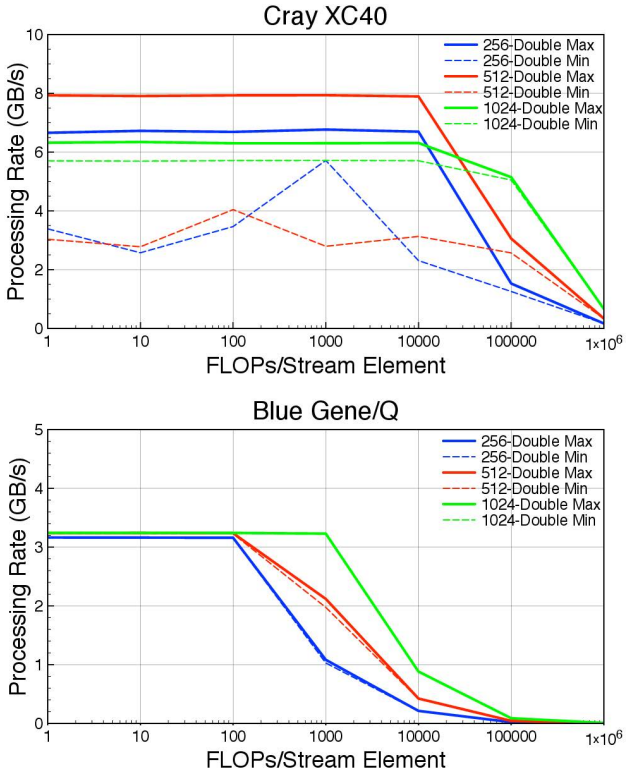

Figure 5: The maximum and minimum processing rate in GB/s varying the number of FLOPs per stream element using the *scale* benchmark with 32 data producers over 32 data consumers. The top panel presents results on a Cray XC40 supercomputer. The bottom panel presents results on a Blue Gene/Q supercomputer.

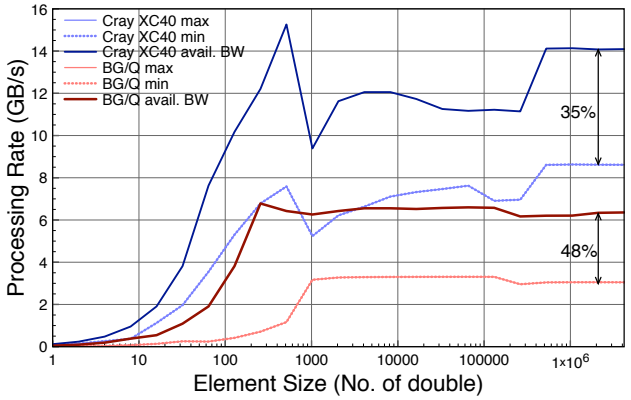

Figure 6: The maximum and minimum processing rate in GB/s varying the size of the stream element using *scale* benchmark with 32 data producers over 32 data consumers on the Cray XC40 and Blue Gene/Q supercomputers. The available bandwidth obtained by running a ping-pong benchmark under the same system configuration of the parallel STREAM benchmark is superimposed on the plot. The gap between the MPIStream library and the available bandwidth is indicated as 35% and 48% on the plot for the Cray XC40 and Blue Gene/Q supercomputers respectively.

served that as the size of stream element increases from a very small size (1 double) to a medium size (256 doubles), the processing rate increases almost linearly on both systems. Beyond the stream element size of 256 doubles, the processing rates approximate a constant limit on both systems.

Figure 6 presents the maximum and minimum processing rate (bandwidth of the MPIStream library) in GB/s varying the size of the stream element on the Cray XC40 and Blue Gene/Q supercomputers. The processing rate increases as the stream element size increases. The maximum processing rate achieved by the MPIStream library is 9GB/s and 3.5GB/s on the Beskow Cray XC40 and Vesta Blue Gene/Q supercomputers respectively. The available bandwidth calculated with *ping-pong* test is superimposed in the plot. The maximum available bandwidth is approximately 15GB/s and 7GB/s on the Beskow Cray XC40 and Vesta Blue Gene/Q supercomputers respectively. The MPIStream library achieves at maximum 65% and 52% of the available bandwidth on the Cray XC40 and Blue Gene/Q supercomputers.

In the fourth test, we studied the potential performance that the MPIStream library can achieve as the number of data producer and consumer scales up. We carried out the test on the Cray XC40 and the Blue Gene/Q supercomputers from 32 data producers over 32 data consumers to 2,048 data producers over 2,048 data consumers. The test used the *scale* benchmark with the stream element of 512 doubles.

Figure 7 presents the maximum and minimum processing rate in GB/s varying the number of data producer and data consumer on the Cray XC40 and Blue Gene/Q supercomputers. The maximum processing rate achieved is 200 GB/s on the Blue Gene/Q supercomputer and 80 GB/s on the Cray XC40 supercomputer. On the Blue Gene/Q supercomputer, we observed a linear increase in the processing rate as the number of data producer and consumer increases. On the Cray XC40 supercomputer, the processing rate increases linearly until 256 data producer and 256 data consumers. From 512 data producer and 512 data consumers, the processing rate starts linear increase again but at a rate lower than the Blue Gene/Q supercomputer.

5. A DATA STREAMING MODEL IN HPC APPLICATIONS

The data streaming model is typically used to solve data-intensive applications. For instance, filtering and map-reduce operations are well-suited to the data streaming model. In addition, it can be effectively used in HPC applications. In this section, we present three use cases of the data streaming model in HPC applications.

5.1 Parallel I/O

The streaming model can be used to address I/O challenges in HPC application, especially the problem of capturing irregular and unpredictable events. We demonstrate the use case in a real plasma simulation code called *iPIC3D*.

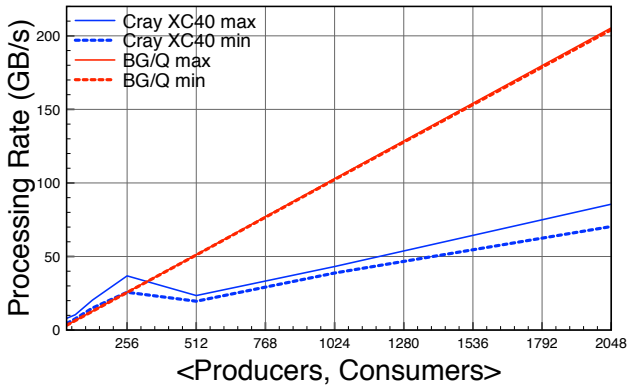

Figure 7: The maximum and minimum processing rate in GB/s varying the number of data producers and consumers using *scale* benchmark on the Cray XC40 and Blue Gene/Q supercomputers.

iPIC3D is a parallel application using computational particles to solve magnetospheric plasma physics problems [8]. Researchers are often interested in saving particles that satisfy certain criteria, e.g. high energy. The distribution of such particles is usually highly uneven on processes and is difficult to predict [15]. Listing 2 outlines the implementation using MPIStream. When MPI processes are updating particle positions and velocities in `ParticlesMover()`, if a particle satisfies the selection criteria, the process switches on the output tag of that particle, streams out the information of that particle and continues with the remaining particles. All particles with the output tag on will be streamed out in the future time steps even their energy might decrease to under the threshold value.

Listing 2: Application for parallel I/O

```
#include "MPIStream.h"
#include "iPic3D.h"
//Intermediate Operation
void PARTICLE_IO(void *in, void *inout, int *len, MPI_Datatype *datatype )
{
    if(*len == 8){//Buffering Data
        double *ptr = (double*)in;
        double *outptr = (double*)inout;
        counter ++;
        memcpy( outptr + (counter - counter/buffersize*buffersize-1), ptr,
                8*sizeof(double));
        if(counter%buffersize == 0){//Flush out data
            MPI_File_write_shared(particlefile,inout,8*buffersize,
                    MPI_DOUBLE,MPI_STATUS_IGNORE);
        }
    }else{
        dprintf("Error: PARTICLE_IO length");
    }
}
//Terminal Operation
void PARTICLE_IO_Term(void *in, void *inout, int *len, MPI_Datatype *datatype )
{
    if(counter%buffersize > 0){
        MPI_File_write_shared(particlefile,inout,8*(counter\%buffersize),
                    MPI_DOUBLE,MPI_STATUS_IGNORE);
    }
     MPI_File_close(&particlefile);
}
int main(int argc, char** argv)
{
     //Initialization
     ...
     //the below is inside KCode.ParticlesMover()
     for (int i = 0; i < KCode.currParticleNum; i++) {
          KCode.updateParticle(particle_list[i]);
          if( isHighEnergyParticle(particle_list[i]) )
              switchOutputTag(particle_list[i]);
          if( isOutputTagOn(particle_list[i]) )
              MPIStream_Send(&(particle_list[i]),&stream);
      }
      MPIStream_Terminate(&stream);
     // Finalization
     ...
}
```

5.2 Threshold Collective Operations

Collective operations are functions that involve all the processes in a communicator. One of its common use is the calculation of the error in an iterative solution of a system. At each iteration, the error is calculated as the sum of the norm of the residual vector on all the processes. If the sum is less than a given error tolerance, the iteration stops. MPI reduction operation with `MPI_SUM` is typically used for this use case. We note that such collective operations can return as soon as the given threshold is exceeded. This could help imbalanced processes, where MPI collective operations impose that the collective operations cannot complete earlier than the slowest processes. The MPI stream library allows us to implement a threshold collective operation that returns early once the threshold value (or other criteria) is violated. Listing 3 presents a possible implementation of such threshold collective operations using `MPIStream_ThresholdColl`. Data consumers receive and sum up the result from data producers. An user-defined operation `Threshold_Op` is defined as the intermediate function to process each stream element. This intermediate function simply sums the received stream elements and compare with the specified threshold value. As soon as the data consumer can draw a conclusion, either the current sum has exceeds the threshold value or all data producers have contributed, the data consumer can return the result to the data producers.

Listing 3: A threshold collective.

```
#include "MPIStream.h"
//Intermediate Operation
void Threshold_Op(void *in, void *inout, int *len, MPI_Datatype *datatype )
{
    int* inptr = (int*)in;
    int* outptr = (int*)inout;
    if(*len==0){
        *(outptr + 0) = 0;
        *(outptr + 1) = 0;
        *(outptr + 2) = 0;
    }else if(*len > 0){
        if( *(inptr) == *(outptr) ){
            *(outptr + 1) = *(outptr+1) +1;
            *(outptr + 2) = *(outptr + 2) + *(inptr+1); //Capture the sum
            //if the result can be concluded now, return result
            if( *(outptr + 1)==NoOfDataproducer || *(outptr + 2)>thresholdValue ){
                result= (*(outptr + 2)>thresholdValue) ?0:1;
                MPIStream_ThresholdColl(NULL, &result, &stream);
                //Move to the next time step
                *outptr       = *outptr + 1;
                *(outptr + 1) = 0;
                *(outptr + 2) = 0;
            }
        }
    }
}
int main(int argc, char** argv)
{
     //Initialization
     ...
     if(isStreamer){
          ...
          //Check the initial solution
          local_result = local_Norm();
          MPIStream_ThresholdColl(local_result, &global_result, &stream);
          ...
          while(!global_result){
              //Start converging solution
              ...
              local_result = local_Norm();
              // End converging solution
              MPIStream_ThresholdColl(local_cnt, &result, &stream);
          }
          MPIStream_Terminate(&stream);
     }
      //Finalization
      ...
}
```

5.3 Online Monitor and Control of Applications

MPI applications divide workload among a group of processes. Process imbalance at run-time could impact the global performance severely [13, 9]. One such example is particle-based applications, in which the particles can move

freely among processes and result in different number of particles on different MPI processes. The MPIStream library can be linked in such applications to provide run-time monitor and control of the applications.
We illustrate a use case in iPIC3D simulating Earth magnetosphere. In such simulation, the processes that handle the inner magnetosphere area could have four times more particles than processes handling other area, even equal number of particles are initialized [14]. For this reason, it is important to monitor the number of particles per process during the simulation. If the workload imbalance reaches a high level, reaction events could be defined, such as redistributing the workload among MPI processes, to mitigate the load imbalance.
Listing 4 outlines the use of the MPIStream library in the iPIC3D code to track if the maximum process imbalance is beyond certain limit. As mentioned earlier, MPI processes only function as data producers when they need to stream out the number of particles processed while at the rest of time they carry on their usual work in the application (move particles and calculate fields). In the example shown in Listing 4, processes stream out the number of particles after they update particles in `ParticlesMover()`. One data consumer keeps on tracking the maximum gap among all processes by calling an user-defined intermediate function `ProcessImbalace()` to update the minimum and maximum number of particles on all processes. If the gap between the minimum and maximum number of particles exceeds a given limit, an alert will be triggered.

Listing 4: Application to monitor and control process imbalance.

```
#include "MPIStream.h"
#include "iPic3D.h"
//Intermediate Operation
void ProcessImbalace( void *in, void *inout, int *len, MPI_Datatype *datatype )
{
    const double gap_threshold = 1.0E8;
    double *min = (double*)inout;
    double *max = (double*)inout+1;
    double *stream_data = (double*)in;
    if(*len == 0)//initialize the results
    {
       *min = DBL_MAX;
       *max = DBL_MIN;
    }else if(*len > 0){
       if( *min > *stream_data) *min = *stream_data;
       if( *max < *stream_data) *max = *stream_data;
     }
     if(*max - *min > gap_threshold) AlertGap();
 }
int main(int argc, char** argv)
{
    //Initialization
    ...
   if(is_data_consumer) // data consumers keep on processing data
       MPIStream_Operate(&stream0);
   if(is_data_consumer){
       iPic3D::c_Solver KCode;
       KCode.Init(argc, argv);
       KCode.CalculateMoments();
       for (int i = KCode.FirstCycle(); i < KCode.LastCycle(); i++) {
           KCode.CalculateField(i); // calculate E field
           KCode.ParticlesMover(); // move particles
           MPIStream_Send(&KCode.currParticleNum,&stream0); //stream out No. of Particles
           KCode.CalculateB(); // calculate B field
           KCode.CalculateMoments(); // calculate moment of distribution function
           KCode.WriteOutput(i); // write file
       }
       MPIStream_Terminate(&stream0); // data producers signal termination of data
   }
   // Finalization
   ...
}
```

In general cases, the MPIStream library can be used to stream online performance data of the application to trigger reactive events. For instance, values from hardware counters can be monitored online, e.g. information on cache misses can be occasionally streamed out to data consumers to check if any performance anomaly occurs on certain processes.

6. CONCLUSIONS

In this work, we investigated the possibility of combining message-passing and data streaming models. We implemented a stream library on the top of MPI, called MPIStream library. We introduced a set of basic functions to support the data streaming model in MPI applications: to globally allocate data producers and consumers (processes to compute on-the-fly results from the stream), to stream continuous and irregular data, to receive and process data and to finalize the streaming operations. In the same spirit as the STREAM benchmark, we developed a parallel STREAM benchmark to measure the data processing rate by the data consumers in the applications using the MPIStream library. Based on the benchmark results, we studied the dependence of the performance on the number of data producers per data consumer, on the computational intensity of processing one stream element and on the size of the stream element. We showed that our MPIStream library can achieve acceptable performance (52%-65% of the maximum available bandwidth). We also demonstrate the potential of our MPIStream library by achieving as high as 200 GB/s and 80 GB/s processing rate using 2,048 data producer over 2,048 data consumers on the Blue Gene/Q and Cray XC40 supercomputers respectively.
The MPIStream library uses persistent communication on the data consumers to lower the overhead of repeatedly posting communication requests of the same argument list. This also mimics the concept of streaming channel. MPI wildcard is used as message source to enable receiving stream elements from any data producer and polling operations are used to process the elements on a first-come-first-serve basis. We use randomised message tag to resolve clashes in message matching and to reduce the contention due to multiple messages matching a single request. We believe that higher processing rate can be achieved in our library by using different implementation approaches, e.g. the Active Access supported by RDMA hardware [1], the Active Message (AM) mechanism [19] and the Pebble Programming Model [20]. It has been showed that an MPI data streaming model can be effectively used to carry out threshold collective operations, to monitor and control applications and to perform parallel I/O of irregular events. These application examples show that it would be advantageous to have native support in MPI for the data streaming model. For instance, MPI could support special message queue that does not require explicitly message matching to effectively implement the concept of streaming channel.
The current work mainly focuses on investigating HPC scientific applications that are based on MPI and can benefit from the streaming model. For the future work, we will survey on more general use cases, e.g. streaming data from the I/O nodes or NVRAM to the compute nodes, linking the library in big-data applications or graph problems. We also intend to investigate using the MPI streaming model to design collective operations that are adaptive to process imbalance at run-time [13].

Acknowledgments

This work was funded by the European Commission through the EPiGRAM project (grant agreement no. 610598. `www.epigram-project.eu`). This research used resources of the Argonne Leadership Computing Facility, which is a DOE

Office of Science User Facility supported under Contract DE-AC02-06CH11357, as well as resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC.

7. REFERENCES

[1] M. Besta and T. Hoefler. Active access: A mechanism for high-performance distributed data-centric computations. In *Proceedings of the 2015 ACM International Conference on Supercomputing (ICS'15)(Jun. 2015), ACM.*

[2] S. Byna, R. Sisneros, K. Chadalavada, and Q. Koziol. Tuning parallel I/O on Blue Waters for writing 10 trillion particles.

[3] W. Gropp and E. Lusk. User's guide for MPE: Extensions for MPI programs. *Argonne National Laboratory*, 1998.

[4] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced features of the message-passing interface.* MIT press, 1999.

[5] T. Hoefler, A. Lumsdaine, and J. Dongarra. Towards efficient mapreduce using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 240–249. Springer, 2009.

[6] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu. DataMPI: extending MPI to hadoop-like big data computing. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 829–838. IEEE, 2014.

[7] E. P. Mancini, G. Marsh, and D. K. Panda. An MPI-stream hybrid programming model for computational clusters. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 323–330. IEEE, 2010.

[8] S. Markidis, G. Lapenta, and Rizwan-uddin. Multi-scale simulations of plasma with iPIC3D. *Mathematics and Computers in Simulation*, 80(7):1509–1519, 2010.

[9] S. Markidis, J. Vencels, I. B. Peng, D. Akhmetova, E. Laure, and P. Henri. Idle waves in high-performance computing. *Physical Review E*, 91(1):013306, 2015.

[10] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. 1995.

[11] R. Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[12] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

[13] I. B. Peng, S. Markidis, and E. Laure. The cost of synchronizing imbalanced processes in message passing systems. In *IEEE Cluster 2014 Conference.* IEEE, 2015.

[14] I. B. Peng, S. Markidis, A. Vaivads, J. Vencels, J. Amaya, A. Divin, E. Laure, and G. Lapenta. The formation of a magnetosphere with implicit particle-in-cell simulations. *Procedia Computer Science*, 51:1178–1187, 2015.

[15] I. B. Peng, J. Vencels, G. Lapenta, A. Divin, A. Vaivads, E. Laure, and S. Markidis. Energetic particles in magnetotail reconnection. *Journal of Plasma Physics*, 81(02):325810202, 2015.

[16] S. J. Plimpton and K. D. Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[17] D. A. Reed and J. Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[18] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[19] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. *Active messages: a mechanism for integrated communication and computation*, volume 20. ACM, 1992.

[20] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *Proceedings of the international conference on Supercomputing*, pages 235–244. ACM, 2011.

[21] R. Williams, I. Gorton, P. Greenfield, and A. Szalay. Guest editors' introduction: Data-intensive computing in the 21st century. *Computer*, 41(4):0030–32, 2008.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[23] X. Zhao, P. Balaji, W. Gropp, and R. Thakur. MPI-interoperable generalized active messages. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 200–207. IEEE, 2013.