

# High Performance Threaded Data Streaming for Large Scale Simulations

Viraj Bhat<sup>1,2</sup>, Scott Klasky<sup>1</sup>, Scott Atchley<sup>3</sup>, Micah Beck<sup>3</sup>, Doug McCune<sup>1</sup>, Manish Parashar<sup>2</sup>

<sup>1</sup>*Plasma Physics Laboratory, Princeton University, NJ {vbhat, sklasky, dmccune}@pppl.gov*

<sup>2</sup>*Department of Electrical and Comp Engr, Rutgers University, NJ parashar@caip.rutgers.edu*

<sup>3</sup>*Computer Science Dept, University of Tennessee, TN {atchley, mbeck}@cs.utk.edu*

## Abstract

*We have developed a threaded parallel data streaming approach using Logistical Networking (LN) to transfer multi-terabyte simulation data from computers at NERSC to our local analysis/visualization cluster, as the simulation executes, with negligible overhead. Data transfer experiments show that this concurrent data transfer approach is more favorable compared with writing to local disk and later transferring this data to be post-processed. Our algorithms are network aware, and can stream data at up to 97Mbps on a 100Mbps link from CA to NJ during a live simulation, using less than 5% CPU overhead at NERSC. This method is the first step in setting up a pipeline for simulation workflow and data management.*

## 1. Introduction

Large scale simulations are increasingly important in many fields of science. The project described in this paper grew from the requirement to deal with the output of a major fusion plasma simulation, the Gyrokinetic Toroidal Code (GTC) [1]. This code examines the highly complex, non linear dynamics of plasma turbulence using direct numerical simulations, and currently generates about 1TB/week of simulation results data during production use.

We have developed a system which efficiently and automatically transfers chunks of data from the simulation to a local analysis cluster during execution. By overlapping the simulation with the data transfer and with the analysis, scientists can analyze their results as they are being produced.

The rate at which fusion scientists generate data from their simulations today is about 1 TB/week, but we expect this figure to increase by an order of magnitude in the next five years. The conventional trend has been to place the generated computational data on the supercomputing sites and later transfer the data manually, or, to execute remote visualization and post-processing of the data. Both approaches encounter difficulty, forcing scientists to concentrate on data transfer and remote visualization issues rather than

dealing with the physics. Remote visualization in particular raises issues of latency and network quality of service. To overcome these challenges we develop a low overhead threaded parallel buffer to transfer data from simulations to the scientist's local computing cluster(s) where access to the data is most convenient and efficient.

The driving force of the threaded buffer for data transfer has been to provide a minimal overhead in simulations while utilizing network resources to the maximum. The application uses simple APIs to activate the transfer. To make this data transfer efficient with the added advantage of global scheduling, optimization of data movement, storage and computation we exploit Logistical Networking (LN) [2] built on the Internet Backplane Protocol (IBP). LN allows for a flexible sharing and utilization of writable storage as a network resource, which is our natural choice for data flow in a data "pipeline" [3] with various depots (storage) locations containing the data in various stages of transformation. The existence of pervasive depots aids in the creation of a reliable data pipelines. It allows simulations to transparently store data to adjacent depots in case of network failures at the receiving end or buffer overflows at the sending end. Post-processing applications can automatically pull/fetch this data through an alternate path from depots adjacent to the computing sites, as the data is transferred from the simulations. This two-way push and pull mechanism enables us to utilize the network bandwidth maximally and affect the simulation's performance minimally.

In this paper we discuss our method of real time data streaming of the simulation data through our threaded buffer, buffer management algorithm, and transformations of the data. Our system creates a high performance data pipeline [4, 5, 6] which enables a more efficient interaction of the scientist with the data. We discuss the various fault tolerant mechanisms used in case of buffer overflow or network failures.

The paper is divided into 7 sections. Section 2 describes Logistical Networking (LN) and the

highlights of using LN in our data transfer mechanism. Section 3 discusses scientific workflow and data pipelines for scientific simulations. Section 4 discusses the threaded buffering scheme; Section 5 elaborates on the working of the threaded buffer with LN and discusses the fault tolerance mechanism in case of buffer overflows and network failures. Section 6 is on results and performance of our buffering scheme. Section 7 discusses future work and conclusions.

## 2. Logistical Networking

Logistical Networking (LN) [2] refers to the global scheduling and optimization of data movement, storage, and computation based on a model that takes into account all of the network's physical resources. Unlike traditional networking, which does not explicitly model storage or computational resources in the network, LN offers a general way of using computing resources to create a common distributed storage infrastructure that can share out storage and computation the way the current network shares out bandwidth. The middleware components that enable logistical networking are arranged in the "network storage stack," [2] analogous to the IP stack, using a bottom-up and layered design approach that provides maximum scalability. Components of the network storage stack are described below bottom up:

**IBP** – Internet Backplane Protocol: IBP is the foundation of the network storage stack and provides a highly scalable, low-level mechanism for managing network storage resources, through shared use of lightweight, time-limited allocations on storage "depots."

**exNode** – External Node: Similar to the concept of an inode in UNIX file systems, this is a generalized data structure which holds the metadata necessary to manage distributed content stored on IBP depots and allow file-like structuring of stored data.

**L-Bone** – Logistical Backbone: Directory and resource discovery service cataloging registered IBP storage depots world-wide.

**LoRS** – **Logistical Runtime System**: The LoRS software suite integrates the underlying capabilities of IBP, the exNode, and the L-Bone into a streamlined tool for storing, accessing, and managing data.

### 2a. Logistical Networking (LN) salient features for Data Transfer

**Data Replication for Fault Tolerance**: The main reason for using the LN is the ability to stream buffers of data (not necessarily entire files) to multiple storage locations simultaneously for fault-tolerance. The

ubiquity of IBP storage means that it is easy to stream data to a number of alternate depots close to the sender and create replicas close to remote receivers. Storing replicas in multiple locations provides fault tolerance in case of network or machine failures. Fault-tolerance through replication is internal to the exNode. The LoRS handles retrieving from multiple replicas automatically [7].

**IBP: Byte Array Abstraction**: We chose IBP as the main transfer mechanism instead of a rigid transfer protocols, because IBP is a more abstract service that is interoperable with a variety of storage resources (disk, ram, etc.). IBP manages blocks of stored data as byte arrays, with details of the storage (fixed block size, differing failure modes, local addressing schemes) masked at the local level. The use of IP networking to access IBP storage resources creates a globally accessible network of storage depots.

### 2b. Selection of LN Technology

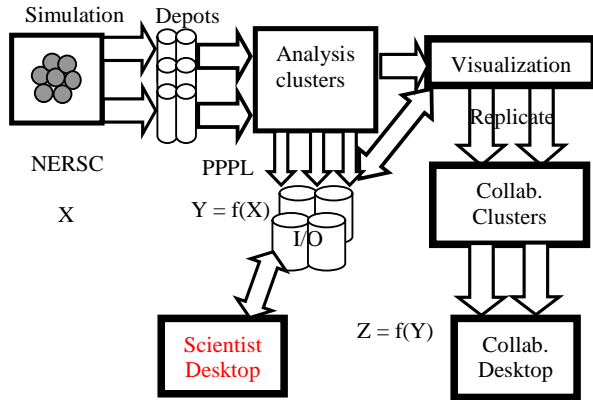
Logistical Networking offers advantages not available elsewhere. Since Grid Protocols [8] do not support replication internally, we would have to use a higher level service such as the Replica Location Service (RLS) to track where copies of the complete files reside [9]. When retrieving the data, we would then have to determine which replica to download. If, on the other hand, we used raw sockets and wanted to implement replication for fault-tolerance, we would have to write our own servers to hold the data, write the transfer management code to use them, and design some method for tracking the replicas and reassembling the pieces—effectively recreating the LN software and infrastructure.

## 3. Data Workflow and pipeline of data simulations

The need for computer aided tools increases with size and complexity of the simulation generating the data. Without automation, Scientists spend a large portion of their time managing the workflow and data flow. Such management includes organizing and sharing raw and derived data between collaborators, transforming data formats, etc.

In this section we would like to illustrate a general data flow pattern of our GTC simulation which runs in parallel on a supercomputer at NERSC, and how this data undergoes continuous transformation until it reaches the desktops of scientists collaborating with PPPL in analyzing the simulation data. We consider

PPPL as one transformation point as it flows along to other collaborators.



**Figure 1: Data pipeline of the GTC simulation**

Figure 1 illustrates the end to end data pipeline used by the GTC simulation running at NERSC. The simulated data is transferred concurrently as the simulation is taking place through our buffering mechanism. The raw data ( $X$ ) streams over to a data analysis center and it is converted into appropriate formats (e.g. HDF5 or NetCDF) as required by the scientists (scientists can specify the format in which they want to transform the data using simple APIs in their codes). The analysis clusters start converting the raw simulation data to an appropriate format for visualization as soon as the first time-step arrives. The converted data ( $Y$ ) is written to disk and fed into visualization routines. This data flow scheme is particularly well suited for the analysis of fusion codes as this makes efficient use of dedicated computing resources at the scientists' local resources and additionally provides the scientists with real-time visualization capability for their simulations. Finally at the end of this data flow, the data reaches the desktops of the collaborators working on the fusion codes who may then further transform the data ( $Z$ ).

#### 4. Threaded Buffer Data Streaming

The goal of the buffering scheme is to transfer data from a live simulation running in batch on a remote supercomputer over a Wide Area Network (WAN) to our local analysis/visualization cluster as efficiently as possible and provide minimal overhead on the simulation [10,11,12]. It should also have replication abilities so that the processed data can be duplicated to collaborators' clusters as and when needed. To avoid loss of raw data either due to buffer overflows (when the generated data does not fit into

the buffer) or network failures, the data should be transferred fault tolerantly.

To achieve this data transfer we use a buffering algorithm that uses a circular queue and a threaded queue manager (one for each node of the super-computer) so that it performs wide-area data transfer with minimal memory overhead on our simulations. This buffering mechanism copies the simulation data to a small memory buffer which is allocated by the user in his/her simulation. The buffer can be thought of as a queue of data blocks expecting to be transferred. This queue is circular, thus it wraps around after it reaches the end. Each data block generated by the user can have varying sizes but the queue manager chops the data into a uniform block size, which is configurable by the user. The queue manager maintains two pointers within the buffer. The first is the *write position*, which is the position where the data is being copied into the buffer (i.e. where the simulation writes data into the buffer). The second is the *send position* to indicate the current position in the buffer where the transfer mechanism is operating (position of last successful transfer). The send position changes in multiples of block sizes. The user can append small pieces of information to the data that contains information for the post-processing routine to operate on data (i.e. metadata). In practice the metadata added to the data never exceeds a small number of bytes and forms a tiny fraction of the actual data to be transferred. The queue manager adds metadata to the data before placing the data on the buffer. The queue manager then updates the values of the *send position* and *write position* whenever data is transferred out of or added to the buffer. After the data is transferred and the *send position* is moved, the application can write into that space. In the next section, we describe the simple buffer management scheme which adapts to the network conditions.

#### 4a. Adaptive Buffer Data Management

We use a simple algorithm to manage the buffer that adapts to both the computation's output rate and network conditions. First, we recognize that the simulation is based on a series of time-steps. The data generation rate is the amount of data generated per step, divided by the time to perform the step. For the GTC code, this can vary from 1 to 90 Mbps, depending on simulation and analysis options.

We also recognize that the network connectivity between the supercomputer and the analysis cluster places an upper limit on the transfer throughput. The

smallest pipe between the supercomputer and the analysis cluster will determine the theoretical maximum throughput for the transfer. Since the transfer routines use TCP for reliable data transfers, we understand that we will get even less than the theoretical throughput [13]. The algorithm tries to dynamically adjust to the data generation rate and the available network rate. It does this by sending all the data that has accumulated since the start of the last data transfer. If the data generation rate exceeds the transfer rate, more data will be in the buffer. In this case, the queue manager will increase the amount of multi-threading in the transfer routines to improve throughput. If the transfer rate exceeds the data generation rate, then less data will appear in the buffer for the next transfer. The queue manager will then reduce consumption of unnecessary network resources. The initial transfer begins after the first time-step is output. All subsequent transfers start as soon as the prior transfer ends.

After some number of time-steps, if the network is stable and the data generation rate is less than the network transfer capacity, then the queue manager tends to reach equilibrium and match the transfer rate to the data generation rate.

Several buffer management states occur, depending on the relationship between data generation and data transfer rates, as is described here:

**A) Data generation rate exceeds transfer rate**

In this state, we maximize the network throughput and move as much of the data to the analysis cluster as possible. In the adaptive buffer transfer mechanism we use the input from the previous step (state) while sending data in the next step and form a loose feedback mechanism. We send the excess data that cannot be transferred to nearby disk and signal the receiving process of this data to start re-fetching this data using any remaining bandwidth, or after completion of the simulation. The queue manager detects this if the simulation needs to write data to the buffer, but the write position is too close to the send position which indicates that there is not enough space in the buffer for the new output.

This makes our scheme “network aware” as our transfers are dependent on the network on which we are operating and the blocks sent out during each transfer depend of the previous transfer.

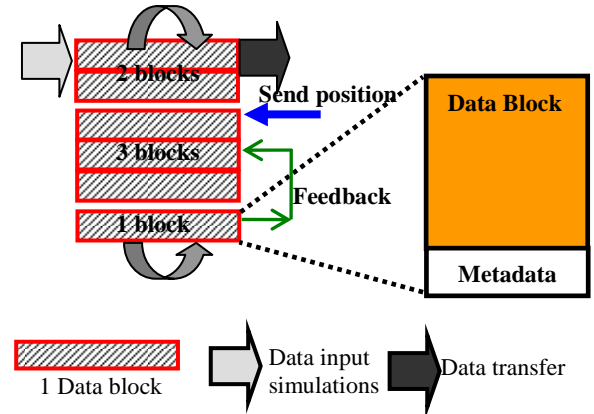
**B) Data generation and transfer rates are similar**

In this situation, significant new data accumulates in the buffer during each transfer. The size of the first transfer is one block. Subsequent transfers usually involve a larger number of blocks. These multi-block transfers use multiple IBP threads and can consume available network capacity.

**C) Data generation rate is small compared to transfer rate.**

If data is generated at a rate in which after every transfer the scheme finds the buffer empty it waits and does nothing till more data is generated in the buffer. In this state the buffering scheme would send out block by block, using minimal network resources.

Figure 2 shows the adaptive buffer management scheme that we use in this paper. This “latency aware” transfer mechanism is particularly useful in cases where blocks are generated quickly around 65-75Mbps as compared to the simple buffer scheme which sends each individual piece in the buffer. It is powerful in cases where data is generated slowly (i.e. less than 1Mbps), in this case if the block size is set to 1MB we send just a single block of data continuously. We believe that that this feedback-based buffer management scheme improves the transfer mechanism by sending as much data as the network can handle and caching the rest to disk until the end of the simulation run. It takes decisions based on the previous transfer when deciding which blocks to transfer and which blocks to write locally. Scheme in Figure 2 works well for transferring data from the simulations at NERSC to PPPL and easily saturates the link as will be shown in the results section.



**Figure 2 Adaptive buffer management scheme**

**4c.Usage of Buffering Scheme**

To take advantage of our transfer mechanism, the application first makes calls to `t_open()`, which initializes a finite buffer and the queue manager. The

queue manager will then wait for any data generated by the simulation. The user then inserts `t_write()` statements at appropriate places in his code where data is generated. The `t_write()` statements copy the generated data to the buffer initialized the user. To close and flush the buffer at the end of the simulation, the application uses `t_close()`. The application can also specify certain information about the data which will be useful for post-processing, by using a `write_metadata()` statement in conjunction with the `t_write()` statements. This statement is useful for starting post-processing at the raw data receiving end. Metadata for the data transfer include global and local dimensions for the global array which will be required for "HDF5" or "NETCDF" or "ASCII" file creation, name of the variables transferred in the data block, name of the final generated file. Metadata size is typically in the order of few hundred bytes.

## 5. Design and Implementation of the Buffering Scheme using LoRS

The design of the streaming mechanism using our circular buffer and queue manager consists of a buffer for each processor on the simulation/computing end which generates data. The threaded write library on the sending end calls the LoRS library which ultimately transfers data using the IBP library to an IBP depot on the receiving end. After the simulation data and its metadata have been transferred, the LoRS library constructs an `exNode` which it returns to the queue manager. The queue manager then sends the `exNode` to a waiting process, `exnodercv`, in the analysis cluster at PPPL via a socket. Although this is an additional step for every transfer, the impact is minimal and provides some benefits. First, each `exNode` does not exceed 10-20KB in size. Second, the `exNodes` (represented as XML) are transferred separately to a program on the receiving end and hence do not interfere with the main data transfer or the computation. Third, since the `exNodes` are represented as in an XML format they allow for platform interchangeability.

The simulations normally run in batch. The receiving part on the PPPL end consists of the `exnodercv` daemon listening for `exNodes` on a well known port. This program keeps track of the data transferred during the simulation and appropriately calls the post-processing routines for visualization/data transformation specified by the user. We have presently incorporated the HDF5 and ASCII routines which generate appropriate files for visualization/post-processing the simulation data. Since the post-processing routines at PPPL read the transferred data

from the depots using the `exNodes` sent to the `exnodercv` daemon, this does not interfere with the running simulation at NERSC. Simultaneously, the post-processing routines can invoke the LoRS augmentation API which replicates the post-processed files and publishes the `exNode` on a well known public web server for later access by collaborators.

### 5a) Failsafe mechanisms

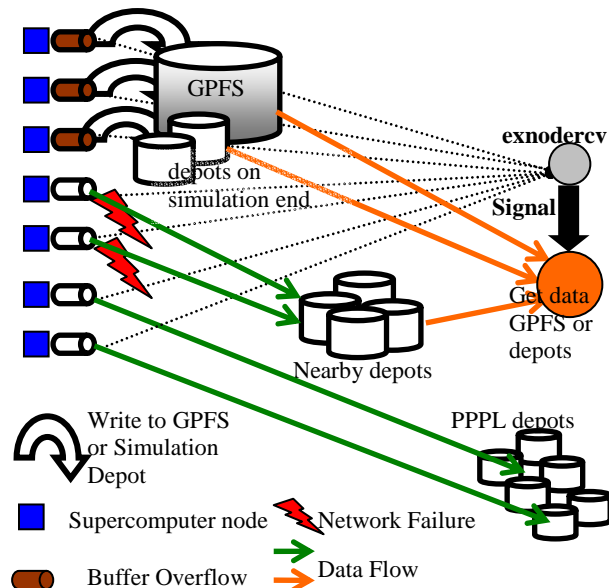
The overall goals of our data transfer mechanism was to provide a low overhead of transfer and fault tolerance. Failures are common in the scenario of the threaded buffer transfer mechanism. The primary causes of failure include:

**1) Buffer overflow at the sending end.** This happens when the data generation rate at the simulation side far exceeds the capacity the network can sustain. This is typically the case when the data generation rate of the program exceeds the maximum network throughput, where the communication time far exceeds the computation time. Presently we are writing the data resulting from buffer overflows which cannot be transferred to our local depots in the form of binary files on NERSC General Parallel File System (GPFS). After the files have been successfully written, a status signal for the failed transfer is sent to the `exnodercv` daemon. The status signal contains the transfer rate, size of failed transfer, and the location of the file to fetch. The daemon program then interprets the status of the failed transfers, like file size and the transfer rate to try to concurrently get the data from GPFS using GridFTP. We would like to be consistent with the transfer mechanism by using LoRS for fetching the failed transfer data written to a local depot (instead of a file written to the GPFS) on the supercomputer, but presently due to security restrictions we are not able to set up a local storage depot on the supercomputer on NERSC.

**2) Network connection to our local depot is temporarily severed.** The LoRS transfer mechanism might be unable to upload data to our local depots due to depot or host failures, lack of storage space, network congestion, etc. To address these issues we upload simulation data to the nearest available depots either on the supercomputer where the simulation is running or on depots located at San Diego Supercomputing Center. We then transfer/write a status/`exNode` generated for these types of upload to our `exnodercv` daemon/alternate depots. Since `exNodes` act as inodes for a network file and contain all replica information (locally and remotely stored), there is no need to separately fetch this data using any special transfer mechanism. The data is fetched from the depots only

during post-processing of the data either during an HDF5, NetCDF or ASCII file creation routines.

Figure 3 illustrates the failsafe mechanism in case of buffer overflows at the simulating end if the data transfer rate can't keep up with the data generation rate. In this case, we write the data to GPFS. We then transfer the status/exNodes which explicitly have an error code for buffer overflow. The exnodercv process uses GridFTP to fetch data from GPFS at the simulation end. It is also possible that the some nodes in the simulation undergo a network failure/timeout. In case of a network failure or timeout of any depots at PPPL, the data is uploaded to the nearest depot using the L-Bone. In our case the nearest reliable depot to the simulation end are the depots at SDSC. We then send the exNodes/status over to our exnodercv process. The analysis processes read these exNodes as usual, but the read performance is less than if the data where written directly to the PPPL IBP depots.



**Figure 3: The Failsafe Mechanisms**

## 6. Results

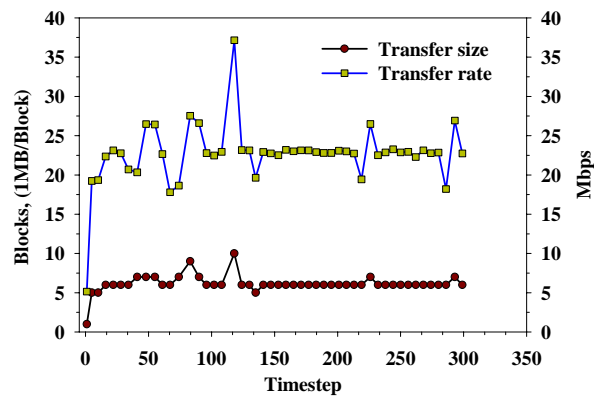
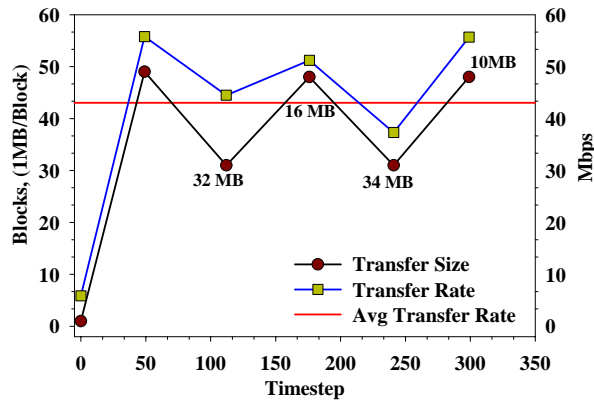
The adaptive buffer management code, which we have developed, is easy to use and has simple APIs which the user can efficiently combine in his simulation to yield a high throughput data transfer. The objective of this work is that the threaded streaming should not slow down the simulation on the supercomputer (i.e. the streaming should add very little to the computation/CPU time).

To evaluate how the data transferred using this buffer and queue manager, we use a sample program that models the GTC simulation which generates

simulation data at every time step. This simulation runs on the supercomputer nodes at NERSC and the data generated is transferred to our local clusters at PPPL. We have employed buffer management with 80 MB buffers per computational node, using 1MB data block sizes. We have used a time-step as the primary reference on the X-axis (each run has 300 time-steps). The Data Generation Rates (Mbps) for each of these experiments is measured by the amount of data generated by the simulation and the time taken to generate them with no I/O involved. Data Transfer Rates is computed by the amount of MB transferred successfully divided by the time taken by the Buffering mechanism to transfer the generated data. We then study the data transfer rate (in Mbps) for various data generation rates which leads to varying data transfer sizes. Buffer overflow corresponds to data written to the local GPFS on the simulation end and must be retrieved by PPPL using the strategies described above. The block size for the transfer is 1MB. Metadata is also transferred along with the data which will be required for post-processing the simulation data. Figure 4a plots the blocks transferred during each timestep and the Mbps corresponding to the blocks transferred. The data generation rate for this experiment is about 320Mbps. Our buffering scheme cannot keep up with this rate, and data is written to local disk in cases where buffer is full (80MB). The buffering scheme initially transfers the first block of data and later sends whatever is remaining in the buffer after transferring the first block. The values at data points correspond to buffer overflows since the maximum data the buffer can hold is 80MB, so when the 49 MB is being transferred data fills the buffer and (63 MB is generated out of which ) 32 MB is written to disk. This process repeats itself until the simulation stops generating data. Thus the data transfer rate is around 43 Mbps. The more data that is in the buffer, the higher the chance for buffer overflow. Figure 4b depicts an interesting case where data is generated at a rate of 21.3Mbps (300MB in 121 sec), all the data generated is transferred without any data written to disk or left un-transferred at the end of the simulation. The buffering scheme starts out with 1 block and then later sends out 6 data blocks but in certain cases where the rate for 6 blocks drop below 20Mbps we transfer around 8 blocks; this leads to oscillations of the data transfer block counts until around 120 timesteps when it reaches an equilibrium of 6 1MB blocks per transfer.

Figure 5 demonstrates the network adaptability of the buffering scheme for a simulation run on two processors. Initially, the data generation rates (20Mbps/Processor) exceed the transfer rates. For each successive transfer, more data is available in the buffer

so the queue manager sends more data and increases the level of IBP threading in the LoRS calls. The buffering scheme stabilizes itself and achieves an overall data transfer rate of approximately 20Mbps.



Figures 4a, 4b: Data streaming with 320Mbps and 21.3 Mbps (single processor). Data points at observed data transfer times.

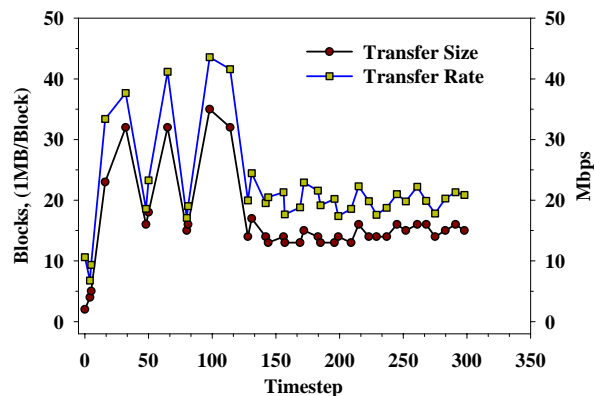


Figure 5: Network aware self-adjusting buffer

Figure 6 shows the high performance buffering scheme which can keep up with rate of generation as high as 85Mbps on 32 processors. All the data generated during this period in the simulation at NERSC is transferred to our local cluster at PPPL. Figure 7 shows significant oscillation due to the higher number

of data generator nodes involved. The best throughput that we can hope to achieve is the minimum of the data generation rate and the theoretical network throughput adjusted for TCP. The data rate is the traffic minus the headers. The maximum traffic from NERSC to PPPL is 100 Mbps, of which we hit 97 Mbps. Thus, this flow used 97% of the link (and all other users got the remaining 3%). The 100 Mbps rate assumes no one else is using the WAN connection so we can expect some value less than 100 Mbps. We can see from Figure 7 how the network can be easily saturated using our buffering scheme. Figure 7 depicts the statistics when the simulation in Figure 6 is operational. It presents an enlarged image of the router statistics. The data rates show that we can achieve a maximum transfer rate of 97 Mbps as shown by a blue spike.

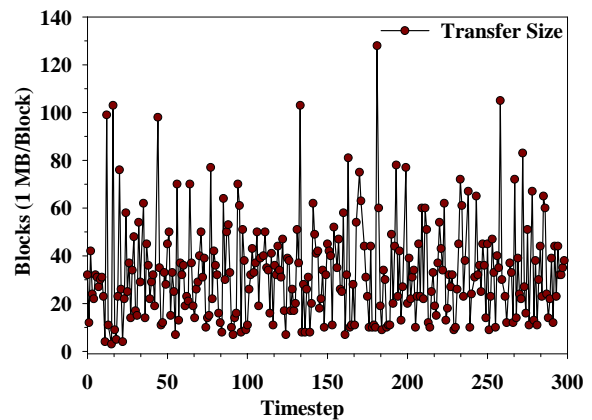


Figure 6: Data Generation Rate of 85 Mbps on 32 nodes (block counts summed over 32 nodes).

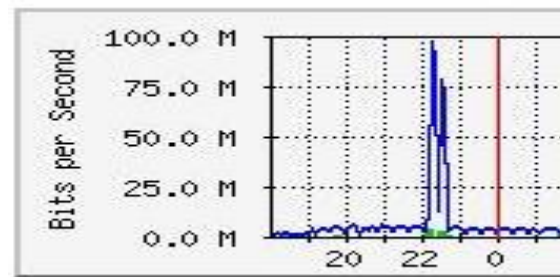
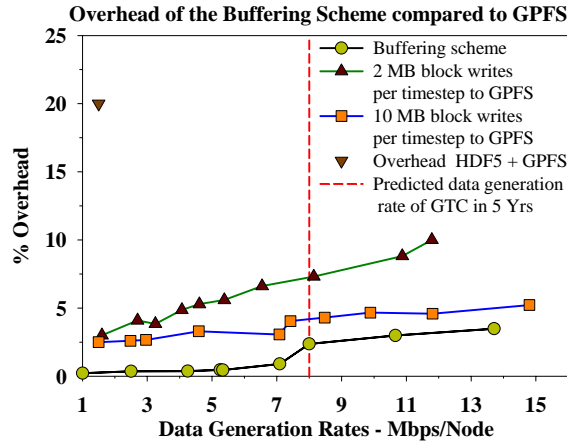


Figure 7: ESNET router, statistics peak transfer rates of 97Mbps/100Mbps at around 22:00. Each data point is a 5 minute average.

Figure 8 shows the overhead of using the buffering scheme with varying Mbps rates and compares this with writing the files to GPFS on the supercomputer nodes. We observe that in cases which are typical for present GTC codes writing data to the GPFS (2Mbps or less per node), overhead is less than for our buffering scheme 5%. In future when the GTC

data generation rates are around 8Mbps, the overhead of using buffering scheme is still small. The present overhead without our buffering scheme (writing to the GPFS at NERSC [1]) is around 20 % when generating hdf5 files.



**Figure: 8. Overhead with Buffering Scheme compared to GPFS (I/O).**

## 7. Conclusions

In this paper we describe development of a threaded mechanism to transfer data with a simple adaptive buffer management scheme for overlapping computation and communication. The buffering scheme had little impact on the simulation with a projected 2% overhead for codes such as GTC running on 1024 processors.

Our scheme adapts dynamically to data generation rates and network throughput, and appropriately adjusts the amount of data transferred and the level of multi-threading to achieve good transfer rates. Our buffering scheme using logistical networking allows for high-performance remote transfer of data with minimal overhead on the computation system. If the data generation rate exceeds the available network resources, we have a failsafe mechanism that uses the available bandwidth to send the bulk of the data while writing the excess data locally and retrieving it later from the remote site.

In the future we will make our fault tolerance mechanism more efficient and take advantage of IBP depots within NERSC. We will work on incorporating our routines into production runs of the GTC code. We have begun working on more optimal MxN [14] mappings for future parallel post-processing modules in our data workflow pipeline. Finally, we will incorporate priority-based transfers for optimized monitoring of selected simulation data output.

## 8. Acknowledgements

This work was supported by USDOE Contract no. DE-AC020-76-CH03073. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. We thank Tech-X Corporation for support in this project.

## 9. References

- [1] S. Klasky et al., "Grid -Based Parallel Data Streaming implemented for the Gyrokinetic Toroidal Code", ACM/IEEE SC2003 Conference, Phoenix, Arizona, USA, November 15 - 21, 2003.
- [2] T. Moore et al., "An End-to-End Approach to Globally Scalable Network Storage", ACM SIGCOMM 2002, Micah Beck, Pittsburgh, PA, USA, August, 2002.
- [3] T. Kosar et al., "Building Data pipelines for High Performance Bulk Data Transfers in a Heterogeneous Grid Environment", Technical Report CS-TR-2003-1487, Computer Sciences Department, University of Wisconsin-Madison, August 2003.
- [4] A. Sim et al., "DataMover: Robust Terabyte-Scale Multi-file Replication over Wide-Area Networks", Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), Santorini Island, Greece, 21-23 June 2004
- [5] "Tools for Creating and Executing Scientific Workflows", <http://seek.ecoinformatics.org>
- [6] I. Altintas et al., "A Modeling and Execution Environment for Distributed Scientific Workflows", 15th Intl. Conference on Scientific and Statistical Database Management (SSDBM), Boston, Massachusetts, USA, 2003.
- [7] J.S. Plank et al., "Algorithms for High Performance, Wide-area Distributed File Downloads", Parallel Processing Letters, (13)2, pp.207-224, June, 2003.
- [8] B. Allcock et al., "Data Management and Transfer in High Performance Computational Grid Environments", Parallel Computing Journal, Vol. 28 (5), pp. 749-771, May 2002,
- [9] A.L. Chervenak et al., "Performance and Scalability of a Replica Location Service", International IEEE Symposium on High Performance Distributed Computing, June 2004.
- [10] X. Ma et al. "Improving MPI-IO Output Performance with Active Buffering Plus Threads", 2003 International Parallel and Distributed Processing Symposium (IPDPS 2003), Nice France, April 22-26
- [11] P.R. Woodward et al. "Distributed Computing in the SHMOD Framework on the NSF TeraGrid", University of Minnesota Computer Science Department: Feb 2004.
- [12] J. Ding et al, "Remote Visualization by Browsing Image Based Databases with Logistical Networking ", SC2003, Phoenix, AZ, USA, November, 2003
- [13] "ESnet Performance Center", <https://performance.es.net/resources.html>
- [14] "MxN Parallel Data Redistribution @ ORNL", <http://www.csm.ornl.gov/cca/mxn/>