Enabling High Speed Asynchronous Data Extraction and Transfer Using DART [*]

Ciprian Docan, Manish Parashar
Center for Autonomic Computing/TASSL Laboratory
Rutgers University, Piscataway NJ 08854, USA
email: {docan,parashar}@caip.rutgers.edu

Scott Klasky
Oak Ridge National Laboratory
P.O Box 2008 Oak Ridge, TN, 37831, USA
email: klasky@ornl.gov

Abstract

As the complexity and scale of current scientific and engineering applications grow, managing and transporting the large amounts of data they generate is quickly becoming a significant challenge. The increasing application runtimes and the high cost of high performance computing resources make online data extraction and analysis a key requirement in addition to traditional data I/O and archiving. To be effective, online data extraction and transfer should impose minimal additional synchronization requirements, should have minimal impact on the computational performance, maintain overall Quality of Service, and ensure that no data is lost.

In this paper, we present DART (Decoupled and Asynchronous Remote Transfers), an efficient data transfer substrate that effectively addresses these requirements. DART is a thin software layer built on RDMA technology to enable fast, low-overhead and asynchronous access to data from a running simulation, and support high-throughput, low-latency data transfers. DART has been integrated with applications simulating fusion plasma in a Tokamak, being developed at the Center for Plasma Edge Simulation (CPES), a DoE Office of Fusion Energy Science (OFES) Fusion Simulation Project (FSP). A performance evaluation using the GTC and XGC-1 particle-in-cell based FSP simulations running on the Cray XT3/XT4 system at Oak Ridge National Laboratory demonstrates how DART can effectively and efficiently offload simulation data to local service nodes and remote analysis nodes, with minimal overheads on the simulation itself.

1 Introduction

High-performance computing is playing an important role in science and engineering and is enabling highly accurate simulations of complex phenomena. However, as the computing systems grow in scale and computational capability, effectively utilizing these platforms and achieving desired computational efficiency becomes increasingly important and challenging. Furthermore, emerging scientific and engineering applications are based on seamless interactions and couplings across multiple and potentially distributed computational, data, and information services. For example, current fusion simulation efforts are exploring coupled models and codes that simultaneously simulate separate application processes and run on different HPC resources at supercomputing centers. These codes will need to interact at runtime with each other, and with services for online data monitoring, analysis, or archiving.

These scientific applications thus require a scalable and robust substrate for managing the large amounts of data generated and for asynchronously extracting and transporting them between interacting components. For fusion simulations, for instance, large volumes and heterogeneous types of data generated have to be continuously streamed from a petascale machine's compute to its service partition, and from there to compute systems that run coupled simulation components, and to auxiliary data analysis and storage machines. A key challenge that must be addressed by such a substrate is getting the large amounts of data being generated by these applications off the compute nodes at runtime, and over to service nodes or another system for code coupling, online monitoring, analysis, or archiving. To be effective, such an online data extraction and transfer service must (1) have minimal impact on the execution

[*]The research presented in this paper is supported in part by National Science Foundation via grants numbers CNS 0305495, CNS 0426354, IIS 0430826 and ANI 0335244, and by Department of Energy via the grant number DE-FG02-06ER54857, and was conducted as part of the NSF Center for Autonomic Computing at Rutgers University. Note that some of the restults in this paper were presented at the Cray User Group Meeting in May 2007, but have not been published in meeting proceeding.

of the simulations in terms of performance overhead or synchronization requirements, (2) satisfy stringent application/user space, time and quality of service constraints, and (3) ensure that no data is lost. On most expensive HPC resources, the large number of compute nodes are typically serviced by a smaller number of service nodes where they can offload expensive I/O operations. As the result, the I/O substrate should be able to asynchronously transfer data from compute nodes to a service node with minimal delay and overhead on the simulation. Technologies such as RDMA allow fast memory access into the address space of an application without interrupting the computational process, and provide a mechanism that can support these requirements.

In this paper, we present DART (Decoupled and Asynchronous Remote Transfers), an efficient data transfer substrate that effectively addresses the requirements described above. DART is a thin software layer built on RDMA technology to enable fast, low-overhead and asynchronous access to data from a running simulation, and support high-throughput, low-latency data transfers. The design, and prototype implementation of DART using the Portals RDMA library [2] on the Cray XT3/XT4 at Oak Ridge National Laboratory is described. DART has been integrated with applications simulating fusion plasma in a Tokamak, being developed at the Center for Plasma Edge Simulation (CPES), a DoE Office of Fusion Energy Science (OFES) Fusion Simulation Project (FSP), and is a key component of a high-throughput data streaming substrate that uses metadata rich outputs to support in-transit data processing and data redistribution for coupled simulations.

A performance evaluation using the GTC [10, 9] and XGC-1 [3] particle-in-cell based FSP simulations is presented. The evaluation demonstrates that DART can effectively uses RDMA technologies to offload expensive I/O operations to service nodes with very small overheads on the simulation itself, allowing a more efficient utilization of the compute elements, and enabling efficient online data monitoring and analysis on remote clusters.

The rest of this paper is organized as follows. Section 2 describes the architecture of DART and its key components. Section 3 describes the implementation on the Cray system and its operations. Section 4 presents the evaluation of DART. Section 5 presents the related work, and Section 6 concludes the paper and outlines future research directions.

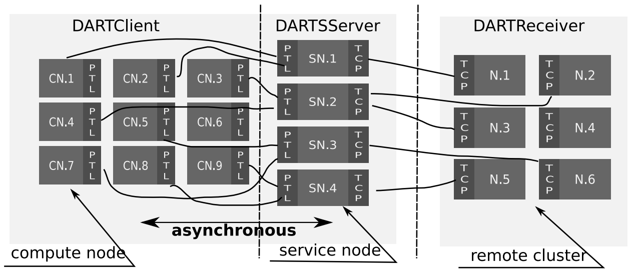


Figure 1: Architectural overview of DART.

2 DART Architecture and Operation

The primary goal of DART is to efficiently manage and transfer large amounts of data from applications running on the compute nodes of a HPC system to local storage or remote locations, to enable remote application monitoring, data analysis, code coupling, and data archiving. The key requirements that DART is trying to satisfy include minimizing data transfer overheads on the application, achieving high–throughput, low–latency data transfers, and preventing data losses.

Towards achieving these goals, we designed DART so that dedicated nodes, i.e., separate from the application compute nodes, asynchronously extract data directly from the memory of the compute nodes using RDMA communication primitives. In this way, we offload expensive data I/O and streaming operations from the application compute nodes to dedicated nodes, and allow the application to progress while data is transferred. We designed DART to provide asynchronous communication abstractions for environments that support only a single thread of execution, i.e., operating systems tuned for scientific computing that do not have support for threads.

DART architecture contains 3 key components as show in Fig. 1: (1) a thin client layer (*DARTClient*) which runs on the compute nodes of a HPC system and is integrated with the application, (2) a streaming server (*DARTSServer*) which runs independently of the application on dedicated nodes and is responsible for data extraction and transport, and (3) a receiver (*DARTReceiver*) which runs on remote nodes and receives and processes data streamed by the *DARTSServer*.

2.1 RDMA Technology

The RDMA communication paradigm provides a window in the address space of a process. It supports process to process communication models with zero-copy, and OS and application bypass. Zero-copy operations can transfer data directly from the user space application memory buffers to a remote destination, and thus it avoids making an extra copy of the data into the kernel space. OS bypass operations can perform data transfers without involving the CPU (e.g., DMA transfers). Application bypass operations can perform a data transfer while an application is running, and without interrupting the application. Thus, RDMA communication mechanisms, are well suited to implement an asynchronous transfer protocol that has a very small impact on a running application.

The Portals [2] library is an abstraction of the RDMA communication paradigm, that provides data transfer primitives for memory to memory communication. It supports transfer operations for extracting messages from a remote node's memory, i.e., the *pull* model ("get" operation) and for injecting messages to a remote node's memory, i.e., the *push* model ("put" operation).

To support memory to memory data transfers between processes, the Portals library maps blocks of an application memory (memory buffers) to the network card, and thus makes them accessible to processes running on remote nodes. To support this mapping, it uses memory descriptors, which are Portals data types containing internal bookkeeping information such as the start address and the length of the memory buffers, current offsets, etc.

The asynchronous data transfers use the application bypass feature, and they require a higher level data signaling and transfer completion notification mechanism. The Portals library can generate software *"events"* for each rdma transfer operation to indicate the start or the completion of a transfer. These events are associated with a specific memory descriptor and Portals uses an internal event queue, that is associated this descriptor, to log them. An application at a higher layer can enable and use these events to implement the required notification mechanism.

2.2 The DART Client Layer

DARTClient is a lightweight software layer that runs on the compute nodes of an HPC resource and is integrated with the application. It is responsible for two key functions, (1) control and coordination and (2) data transfer.

The control and coordination function consists of registering a compute node with a streaming server at startup, and posting requests for transfer notifications to the streaming server at runtime. During the registration phase, the DARTClient layer at the compute node and the streaming server exchange identification and communication parameters such as unique numerical instance identifiers, communication offsets and memory descriptor identifiers.

The data transfer function consists of setting up the communication parameters, waiting on transfers completion notification and releasing resources. To post a transfer request, the DARTClient layer at the compute node notifies the streaming server that it has data ready to be sent, and sets up the parameters and priorities for the transfer. It is then the responsibility of the streaming server to actually schedule, execute the transfer of the data, and notify back the DARTClient on the completion of the operation.

We build the DARTClient directly on the RDMA mechanism to expose two classes of memory descriptors to the streaming server. The first class consists of application level memory descriptors that DARTClient creates during the registration phase, announces to the associated streaming server, and preserves throughout the lifetime of the application. DARTClient defines two types of application level memory descriptors – one is used to post request for transfer notifications, and the other is used to retrieve transfer completion acknowledgments from the streaming server.

The second class consists of stream level memory descriptors that DARTClient creates for each data stream the application generates. A data stream is a continuous flux of bytes that have the same destination, e.g., a file. Stream level memory descriptors are also of two types – the first is used to set up communication parameters and signals associated with a data stream, and the second is used to map and expose the actual memory buffers for the streaming server to fetch and transfer from the application. DARTClient creates these memory descriptors on-the-fly when the application generates a new data stream, and announces them to the streaming server as part of the notification message.

DARTClient provides asynchronous abstractions that enable the application layer to make non-blocking "send" calls, and continue its computation without waiting for the transfer to complete. The underlying mechanisms use OS/application-bypass capabilities provided by RDMA transfers to overlap data transfers with application computations, and mask the cost of data transfers. DARTClient uses multiple memory buffers to allow multiple data streams of different sizes to proceed in parallel.

The client layer, also supports a cooperative data transfer mode, in which multiple compute nodes write in the same data stream, e.g., file. In the cooperative mode, DARTClient is flexible and allows the application to define and group together the nodes that write to the same data stream. Based on the groups defined at the application level, DARTClient coordinates the nodes in a group at runtime, and for each node computes an offset in the common stream where the node should write at. DARTClient includes this offset in the notification message that is post to the streaming server, and allows the server to be stateless with respect to a data stream.

2.3 The DART Streaming Server and Receiver

DARTSServer is the DART streaming server component that runs as a standalone application on dedicated nodes of a HPC system, and is responsible for asynchronously extracting data from applications running on the compute nodes, and streaming it to remote clusters or to local storage. Multiple instances of the DARTSServer can run on different nodes, and cooperatively can service data transfer requests from the larger number of compute nodes.

The key functions that DARTSServer provides include registration of the communication parameters received from a compute node or another instance of the DARTSServer, waiting for notification posts of data transfer availability from the compute nodes, scheduling and managing data extraction from the compute nodes, and transferring the data out to local storage or to remote clusters as requested by the compute nodes. Data transfers have priorities based on the size of the data blocks and the frequency at which the application generates them. DARTSServer assigns a higher priority to smaller and more frequent data blocks, and assigns a lower priority to the less frequent and larger data blocks. This priority mechanism prevents transfer of smaller blocks from being blocked by the transfer of larger blocks and enables them to proceed in parallel with the larger block transfers.

The primary performance goal of the DARTSServer is to minimize transfer latencies and maximize transfer throughput. To achieve these goals, the streaming server pre-allocates communication memory buffers and associates them with memory descriptors at startup. The number of pre-allocated memory buffers and the size of each buffer is influenced by multiple factors such as (1) the maximum limit on the number of file descriptors that can be used by a process (a memory descriptor counts as an open file), (2) the memory available on the host node, and (3) the size of the working set (to avoid trashing behaviors). For example, the implementation on the Cray XT3/XT4 at Oak Ridge National Laboratory (described in more detail in Section 3) allocates 512 memory descriptors, and a buffer of 4 MB for each descriptor.

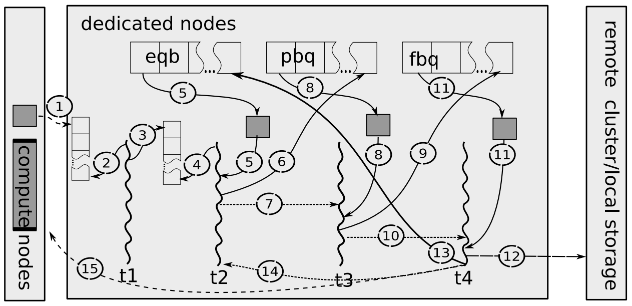


Figure 2: Data flow describing the operation of DART.

In addition to the memory descriptors used for the data transfers, the streaming server also allocates memory descriptors for compute node registration and transfer signaling. The transfer signaling function uses two descriptors on the streaming server – the first descriptor accepts notification posts from compute nodes of data availability, and the second descriptor is used to send notifying acknowledgments to compute nodes when an asynchronous transfer completes.

The DARTSServer is multi-threaded to allow asynchronous and parallel data streaming, e.g., fetching new data from compute nodes while streaming previous data to a remote cluster, as well as the data transfer scheduling management to progress simultaneously, as illustrated in Fig. 2. The server uses multiple queues to (1) manage the incoming compute nodes transfer requests, (2) to schedule and prioritize streaming operations, (3) to track and manage the memory buffers in different states of a transfer, and (4) to reduce synchronization contention between the server threads. The server caches the memory buffers and descriptors, and links them in different queues according to the status of a data transfer, e.g., *eqb* contains unused memory buffers, *pqb* contains memory buffers for in-progress data transfers, and *fqb* contains used buffers that are ready to be streamed or saved.

The overall data flow associated with the operation of the DARTSServer is illustrated in Fig. 2. A data transfer starts with a request for send message generated by the application on a compute node and posted on the notification descriptor (1) of the server. The server extracts the notification messages in the order of arrival (2), and encodes and inserts them in a priority queue accordingly (3). Then, the server processes the priority queues (4), schedules the data transfers (5 and 6) and starts the asynchronous data extraction from the compute nodes (6 and 7). A different thread in the server monitors the asynchronous data completion (8), and schedules the data streaming or saving (9 and 10). Then, the server finishes a transfers (11, 12 and 13) and posts an acknowledgment notification message on the notification memory descriptor of the corresponding compute node.

2.4 DART API and Usage

The DART client layer (DARTClient) provides I/O primitives that are very similar to standard Fortran file operations, i.e., *dart_open(), dart_write(), and dart_close()*. We chose the names and signatures of these operators to make the library easy to use by scientists and to be incorporated into existing application codes. DART additionally provides functions to initialize and finalize the DART library, i.e., *dart_init()*, and *dart_finalize()*. DART also provides API for compute node coordination to enable cooperative operations and interface with additional middleware.

DART operators are asynchronous. Typical usage of the DART API consists of a stream open *dart_open()* operation that returns a handle, one or more non-blocking *dart_write()* send operations on this handle, and a stream end *dart_close()* operation on the handle. A DART operation may be in one of three states, *"not started"*, *"in progress"* or *"finished"*. In case of the first two states, DART maintains stream state information, and when the application calls *dart_finalize()*, it ensures that all in-progress operations finish. As a result, a call to *dart_finalize()* can block and should be called at the end of the application so that it does not impact the simulation.

3 Implementation of DART on the Cray XT3/XT4

3.1 System Description

We implemented DART on Jaguar, the Cray XT3/XT4 at Oak Ridge National Lab. Jaguar [1] has 11k nodes logically divided into compute and service nodes.

XT3 is the older version of the machine, and is built up from compute nodes with dual-core 2.6GHz AMD Opteron CPUs, and 4GB memory per node, i.e., 2GB per core, and runs Cray's UNICOS/lc operating system with the Catamount micro-kernel [7]. The operating system on the compute nodes is specially tuned for scientific computing. To meet the scientific computation performance requirements, the OS removed some features such as the sockets interface, the TCP and UDP transport protocols, the shared memory support, or the multi-threading support. These tunings impose additional constraints and challenges on the implementation of a fast and asynchronous I/O substrate. A service node has a 2.4 GHz single-core AMD Opteron CPU with 4GB memory and runs a full fledged Linux kernel.

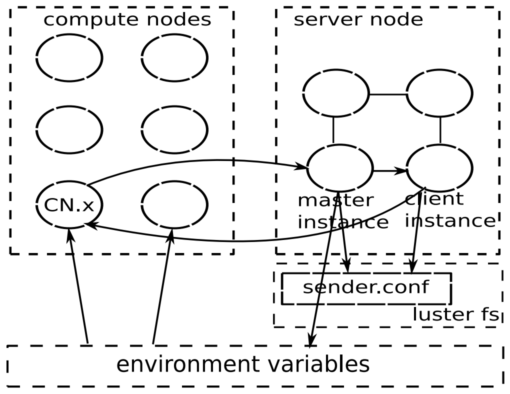


Figure 3: DART compute node bootstrap mechanisms.

XT4 is the newer version of the machine, and is built up from compute nodes with quad-core 2.1GHz AMD Opteron CPUs, and 8GB of memory per node, i.e., 2GB per core, and runs Compute Node Linux (CNL) operating system. CNL is a version of the Linux kernel, that is customized for scientific computing environments. In contrast with Catamount, CNL scales better to quad-cores and adds support for multi-threading, however it still does not provide sockets interface or higher level transport protocols such as TCP or UDP. The service nodes have 2.6GHz dual-core AMD Opteron with 8GB of memory.

Jaguar's compute and service nodes are interconnected in a 3D Torus topology by Cray Seastar routers through the proprietary *HyperTransport* interconnection bus that is able to deliver a transfer bandwidth of 6.4GB/s. The Jaguar service nodes are connected to a remote cluster, i.e., Ewok, by a 5GB/s aggregated link. The Ewok cluster has 128 nodes, each having 2 Intel Xeon 3.4GHz CPUs and 4GB memory. The nodes are interconnected by an InfiniBand switch through a 1GB/s link. In our implementation, DARTClient runs on the Jaguar compute nodes, DARTSServer runs on the service nodes and DARTReceiver runs on the Ewok nodes.

3.2 Compute Node Bootstrap Mechanism

On Cray XT3/XT4 machine, compute nodes communicate with service nodes using Portals RDMA calls. Portals identifies each process in the system by a Portals address, which is a tuple composed of a unique numerical identifier for the node (compute or service) and for the process, i.e., *"nid"* and *"pid"*. Two processes in the system can communicate, if they know each other Portals address.

To run a simulation job in the system, a user requests a certain number of compute nodes for the job. However, the specific nodes assigned to the simulation, and consequently their Portals addresses, are determined only at runtime. The scientific codes running on the compute nodes and the streaming server are two independent applications, and to enable them to communicate over Portals and using RDMA calls, we define a customized bootstrap mechanism that is independent of the underlying execution mechanism (e.g., MPI or the batch queuing system).

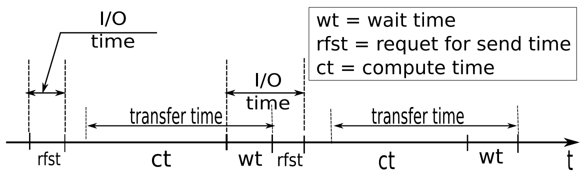
The architecture of our bootstrap mechanism requires that the streaming server is running at the time when the application starts, and relies on a shared file-system between the nodes that run the instances of the streaming server. The bootstrap mechanism works as follows (see Fig. 3) the first instance of the streaming server writes its Portals address (i.e., the tuple *"nid"*, *"pid"*) to a persistent configuration file, and also exports this values into environment variables. When the application initializes, DARTClient parses the corresponding environment variables and reads the Portals address of the streaming server. In turn, DARTClient sends its own Portals address to the streaming server. Note that this bootstrapping approach allows the streaming server to restart, get a different address and modify the configuration file and the environment variables multiple times, before the application starts on the compute nodes (which happens after the application is launched through a batch queuing system).

In case of multiple streaming server instances that run on different nodes, the first server with the lowest "nid" value acts as a master and locks and writes the configuration file. The other streaming server instances check the existence of the configuration file and read the master instance Portals address, then they register their own Portals addresses with the master instance. During the application initialization phase, the master server maps each compute node to an instance of a streaming server via the bootstrap process, and later the compute node forwards all its requests to that instance of the server. In this process, the master server uniformly distributes the compute nodes across the streaming servers, so that each server serves approximately the same number of compute nodes.

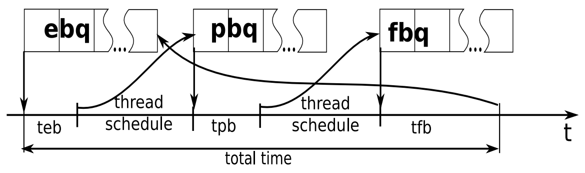
In the implementation of DART described in this paper, the initial mapping of compute nodes to servers is preserved for the runtime of the applications. This is because the load on the servers is naturally balanced due to the SPMD (single program multiple data) nature of the applications. Cooperative load balancing between streaming servers is used in cases where this is not true (e.g., if local adaptivity is used).

3.3 Data Transfer Protocol

As described in Section 2, DARTClient is lightweight and implements the bare minimum functionality so as to minimize its impact on the application. Consequently, the DARTSServer implements all the logic associated with scheduling, data extraction and transfer. It does this using a two step protocol. In the first step, the compute node posts a small *"request for send"* notification message into the memory descriptor of the streaming server instance it is associated with. In the second step, the streaming server schedules the request, extracts the data from the memory of the compute node, and transfers the data to the destination specified in the *"request for send"* notification (i.e., remote node or local storage). Multiple data notification requests from the same or multiple compute nodes are serviced in a round robin fashion according to the priorities of the requests.



(a) DART client.



(b) DART steaming server.

Figure 4: Timing diagrams illustrating the operation of *DARTClient* and *DARTSServer*.

We illustrate the operation of DART on the Cray XT3/XT4 system using timing diagrams presented in Fig. 4. A usual run of a scientific application consists of sequences of computations (identified by c_t in the Fig. 4(a)), followed by data transfer requests (identified by w_t and rfs_t in the Figure). The data extraction protocol is asynchronous, and once the compute node sends a *"request for send"* notification (i.e., rfs_t), it can continue its computations without waiting for the actual transfer to complete. Although the data extraction may not start immediately, it can proceed in parallel with the computation. A compute node may block before sending the notification message (i.e., w_t), if the streaming server is busy and has not yet processed a previous request from the that compute node. The effective data transfer time at the application level is the sum of w_t and rfs_t times.

In Fig. 4(b), we present a timing diagram for the transfer of a block of data from a compute node to a remote node. In the Figure, *"teb"* is the time the server takes for processing a notification message from a compute node and scheduling the corresponding data extraction, *"tpb"* is the time the server takes for extracting the data from the compute node using the Portals interface, and *"tfb"* is the time the server takes to transfer the data to a remote node using the TCP interface. The value of these parameters corresponds to the processing time of the threads $t_1 + t_2$, t_3, and t_4 as described in Section 2. The value for the total I/O time equals the sum of these three values plus some OS thread scheduling overheads.

4 Experimental Evaluation

We evaluate the performance of DART using two sets of experiments. The first set evaluates the base performance of DART using synthetic simulations. The synthetic simulations follow the same computational pattern as the real simulations, i.e., a set of one or more computational stages followed by one I/O stage. The experiments evaluate the core parameters that influence data extraction rates from compute to service nodes. This includes (1) the size of data unit fetched, i.e., the block size, (2) the number of memory descriptors allocated at the service node, (3) the frequency of data fetches, and (4) time required to stream the data, i.e., the time to store the message at the destination location.

The second set of experiments evaluates the performance of DART when it is integrated with real simulation codes, i.e., the GTC and XGC-1 fusion simulations. The key metrics used in these experiments are (1) the effective data transfer rate and (2) the overhead of using DART on the performance of the simulation.

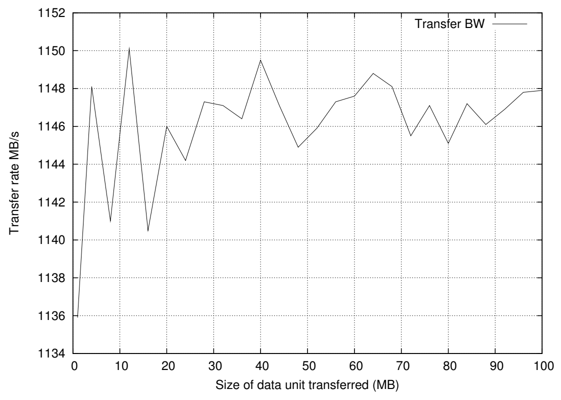


Figure 5: Data transfer rate between compute and service nodes.

4.1 Data Transfer Rate

In this experiment, we test the maximum application-level transfer rate that we can achieve between compute and service nodes, using RDMA data transfers.

We set up this experiment using two compute nodes and one service node, i.e., one streaming server extracts data from a simulation running on two compute nodes and measures the transfer rates. The synthetic simulation used in this test has a compute stage of 1 *msec*. This parameter however does not not influence the transfer rates since we measure the transfer times and sizes at the streaming server.

In this test we vary the size of the data unit (i.e., the block size) from 1 to 100MB in increments of 4MB, and measure the achieved transfer rate for each value. We run the synthetic simulation for 100 iterations, and perform 100 data transfers for each value of the block size. We plot the average transfer rate (over the 100 iterations) for each block size in Fig. 5.

The plot shows that we can saturate the communication link between compute and service nodes using a minimum block size of 4MB. The saturation value is achieved at application level, and represents the maximum remote memory to memory data transfer rate that the system can effectively sustain. Note that we use the `gettimeofday()` system call to measure the transfer times, and as a result, the measured values also include system operations such as OS process scheduling, etc., which explains the variability in the rates measurements.

4.2 Efficient Computations and Transfers Overlapping

In this experiment we test the influence of the compute stage duration and the frequency of I/O on the efficiency of computation and data transfer overlapping.

We set up the experiment using 128 compute nodes running the synthetic simulation and sending data to one instance of the streaming server. On the compute node side, we use memory buffers of 4MB, and on the streaming server side we use 128 memory descriptors to ensure that we can fetch the memory buffers from all the compute nodes in parallel. Also, we allocate 4MB of memory for each descriptor to match the size of the memory buffer fetched. Further, the streaming server transfers the data to a node on a remote cluster. We varied the duration of the compute stage for the synthetic simulation to control the frequency of the I/O operations.

The results for two tests in this experiment are presented below. The duration of the compute stages for the synthetic simulation in these tests is selected to demonstrate the effect of I/O frequency on the application overhead, and on the ability to overlap I/O with computations.

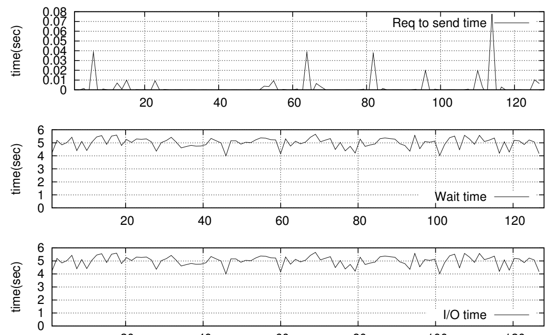
Note that the synthetic simulation posts the "request for send" notifications from all compute nodes at the same time, while the streaming server processes these notifications sequentially in round robin fashion. The streaming server can service the $i+1^{st}$ notification request from a compute node only after it finishes serving all i^{th} requests from the other compute nodes. While serving a transfer request, the most time consuming operation at the streaming server is to stream the data to the remote location.

We derive the formula for the upper time value that the server needs to service two consecutive requests for the same compute node as $t_{fb} * N + t_{pb}$, where t_{fb} represents the time required to transfer the content of a memory buffer to a node in the remote cluster (Ewok), N is the number of compute nodes, and t_{pb} represents the additional time to process and fetch the data from the compute node.Note that the value for t_{pb} is typically negligible as compared to $t_{fb} * N$.

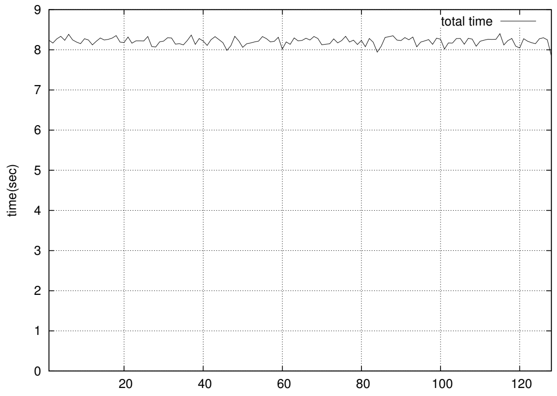
Our goal is to try to find a good balance between the duration of a compute stage and the data size streamed, to better overlap computations with data transfers.

In each of the tests, we measure the *I/O time*, the *wait time* and the *"request to send" time* at the compute nodes and the total *I/O time* for an operation (from the "request to send" notification to the end of streaming) at the streaming server.

In the first test we set the duration for the compute stage to 2 *sec.* We choose this value based on previous experiments to demonstrate the effects of poor overlap. The results from this test are plotted in Fig. 6(a) (compute node) and Fig. 6(b) (streaming server). In these plots, the x axis is the numerical identifier of each compute node and the y axis is the cumulative I/O time for 100 simulation iterations. The I/O time on the compute nodes equals the sum of "request to send" and wait times.



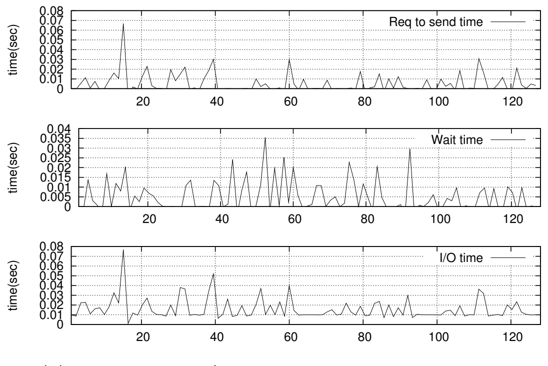
(a) Cumulative I/O time at the compute nodes.



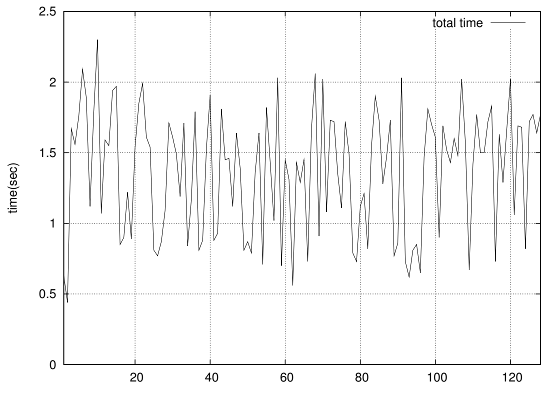
(b) Cumulative I/O time at the streaming server.

Figure 6: Overhead of DART on the application for a compute phase of 2 *sec.*

The values for wait time and I/O time in Fig. 6(a) are almost identical, and this indicates that the I/O time is dominated by the wait time. A large value for the wait time indicates that a notification operation blocks, and waits for a previous operation to finish. In this test, the duration for the compute stage is small, and the frequency of I/O stages is higher than the streaming server's ability to transfer the fetched data to the remote location (i.e., a node on the Ewok cluster). As a result, the streaming server delays the processing for the $i+1^{st}$ notification from compute nodes and this causes the large I/O time in Fig. 6(b). The results of this test show that the overlap between computations and the overhead of data extraction is not optimal, as the wait time is very high on compute nodes. The average I/O time difference for a memory buffer transfer as measured on the streaming server and on the compute nodes is 34 *msec*, and represents the additional time the streaming server spends to transfer the data from a compute note to the remote node.



(a) Cumulative I/O time at the compute nodes.



(b) Cumulative I/O time at the streaming server.

Figure 7: Overhead of DART on the application for a compute phase of 4.3 *sec*.

In the second test, we set the duration for the compute stage based on the upper limit formula $t_{fb} * N + t_{pb}$ established above and the results from the previous test to be 4.3 *sec*, where $t_{fb} = 34$ *msec*, $N = 128$, and we neglected the t_{pb} term. The rest of the setup and parameters remain the same.

The results of the test are plotted in Fig. 7(a) for the compute node and in Fig. 7(b) for the streaming server. Again, in these plots the x axis is the numerical identifier of each compute node and the y axis is the cumulative I/O time for 100 simulation iterations. The value for the wait time in this case, is very small (5 *msec*), indicating that the $i + 1^{st}$ fetch operation from a compute node can proceed almost immediately, without having to wait for the i^{th} transfer to finish. These results demonstrate an efficient overlap of simulation computations with data extractions, resulting in very low I/O overhead on the simulation.

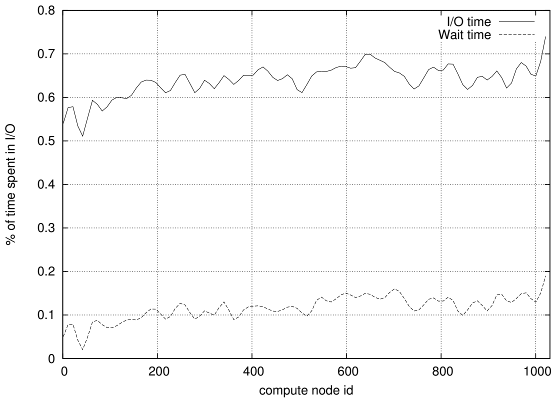
These experiments demonstrate that the data transfer latency and the simulation overhead depend very strongly on the duration of the computation stage (i.e., the frequency of I/O operations) and the size of the data blocks. **We would like to mention that we can not always directly control and tune these parameters for the best results, e.g., in real scientific codes the physicists do not have precise mechanisms to set the values for these parameters, nor do they have the knowledge to do so. However, we believe that understanding the parameters that influence a data transfer allows us at least to increase the number of streaming server instances that service the I/O requests of an application and keep a low I/O overhead on the application.**

4.3 Integration with Simulation Codes

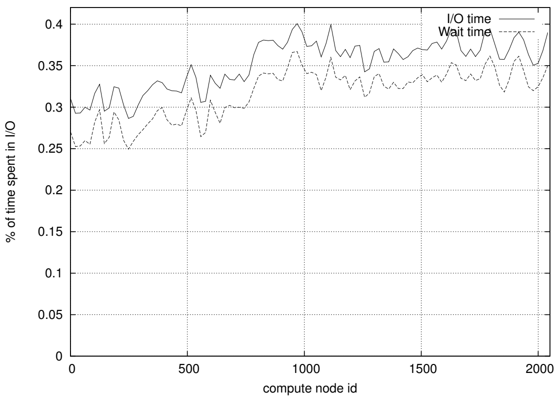
In the second set of experiments, we test the additional overhead that DART operations impose on a simulation running on the compute nodes, and express the overhead as a percentage of the time the simulation spends in its compute stages. For these experiments, we integrate DART with real scientific application codes and analyze its performance using multiple criteria. First, we analyze the throughput of the streaming server on both, the Portals (compute nodes to streaming server) and TCP (streaming server to remote node) interfaces; second, we analyze the I/O overhead on the simulation, and third, we analyze the scalability of DART.

Integration with GTC: In this set of experiments, we use the Gyrokinetic Toroidal Code (GTC) simulation. GTC is a highly accurate fusion modeling code that performs a highly complex analysis of non linear dynamics of plasma turbulence by using numerical simulations, and requires very-large scale computing capabilities. We couple DART with the GTC code and vary simulation parameters in the runs described below. The key GTC parameters that influence the size of the data generated and the frequency of I/O are *micell, mecell*, which determine the size of the global particle domain, *npartdom*, which determines the number of particle sub-domains, and *msnap*, which determines the frequency of I/O, i.e., the number of restart file produced.

To test the scalability of DART, we run the GTC simulation on 1024 and 2048 compute nodes. For the test on 1024 compute nodes, we set the *micell* and *mecell* parameters to 160, *npartdom* to 16, and *msnap* to 13. With these parameter values, the GTC code generates a checkpoint of 40MB every 13 simulation iterations. Each simulation iteration takes 8 *sec*, and the simulation runs for 200 iterations. Proportionately, for the test on 2048 compute nodes, we set the *micell* and *mecell* parameters to 160, *npartdom* to 32, and *msnap* to 26, and ran the GTC code for 400 iterations. Using these parameters, the GTC code generates a checkpoint of 40MB every 26 simulation iterations, and each iteration takes 8 *sec*. In these experiments, we run the streaming server on a dedicated service node (i.e., no other users or I/O services are running on the same node). To analyze the performance of DART, we measure the *wait* and *I/O times* at the simulation level for each of the compute nodes, and the throughput obtained on both Portals and TCP interfaces at the streaming server. During these experiments, the remote receiver receives the data from the streaming server, but does not perform any processing on the data.



(a) I/O overhead on the GTC code on 1024 nodes.



(b) I/O overhead on the GTC code on 2048 nodes.

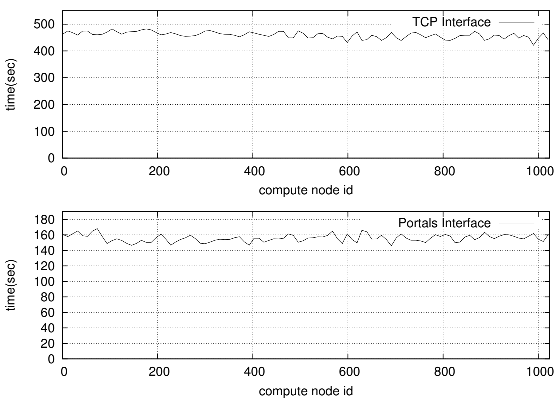
Figure 8: Percentage of I/O and wait time overhead on the GTC simulation measured at the compute nodes.

The I/O overhead on the GTC application caused by DART data extraction and streaming is plotted in Fig. 8 as a percentage of the time taken by the compute stages. The x axis in these plots is a numerical identifier for each compute node, and the y axis is the overhead measured at each node of the simulation.

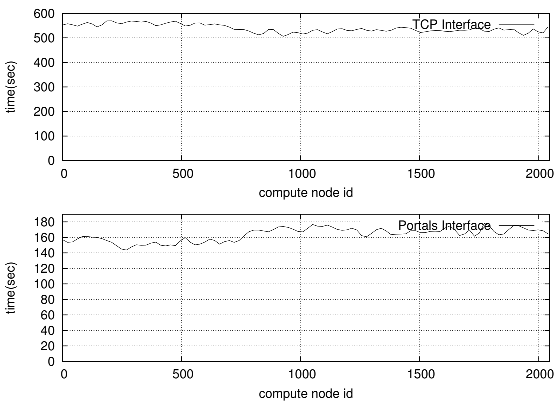
In Fig. 8(a), the average value of the percentage *wait time* across the 1024 nodes is $< 0.15\%$, which indicates that DART is able to efficiently overlap computations with data extraction and transport. Further, the average value of the I/O time is $< 0.65\%$, i.e., the overhead on the simulation is very small. In Fig. 8(b) the average value for the *I/O time* for 2048 nodes is still small at $< 0.4\%$. However the difference between the *I/O* and *wait time* percentages is much smaller than in the 1024 case, indicating that the streaming server is reaching the limit for the maximum number of compute nodes it can service simultaneously. The average value for the percentage overhead for the test on 2048 nodes is smaller than that for the test on 1024 nodes because the GTC code is generating the same amount of data for its checkpoints, but the length of computing stage is doubled.

Fig. 9 plots the cumulative *I/O times* at the streaming server for (1) extracting the checkpoint from the simulation using the Portals interface, and (2) to transport the data to a remote node using TCP interface, for each compute node over the runtime of the simulation. In these plots, the x axis is a numerical identifier for each compute node and the y axis is the cumulative *I/O time*. As expected, the transfer times for the Portals interface are smaller. The corresponding values for the transfer time (both Portals and TCP) are similar for the two cases (1024 and 2048 compute nodes), because the buffers that the streaming server allocates for communication are filled to capacity throughout the tests.

Before running these tests, we measured the link throughput between the streaming server and the remote node using the TCP transport protocol, and the peak measured value was 5.01 *Gbps*. In Fig. 10, we plot the throughput achieved using DART at the streaming server on both, the Portals and TCP interfaces. In this plot, the x axis is the runtime of the simulation, and the y axis is the throughput as a percentage of the peak value. In both, the 1024 and 2048 node tests, we achieve on the TCP interface an average sustained throughput of 60% of the peak TCP throughput. The rest of the time at the streaming server is spent on Portals transfers and the scheduling of the extract operations. The average throughput on the Portals interface is comparable to the value of the TCP interface, i.e., 60%, which is expected since the communication buffer at the streaming server is fully

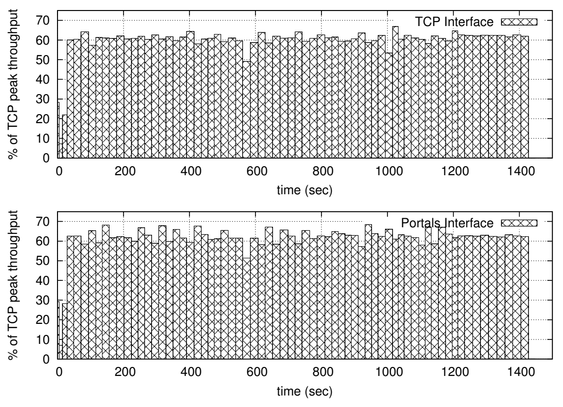


(a) Cumulative I/O time at the streaming server servicing 1024 compute nodes.

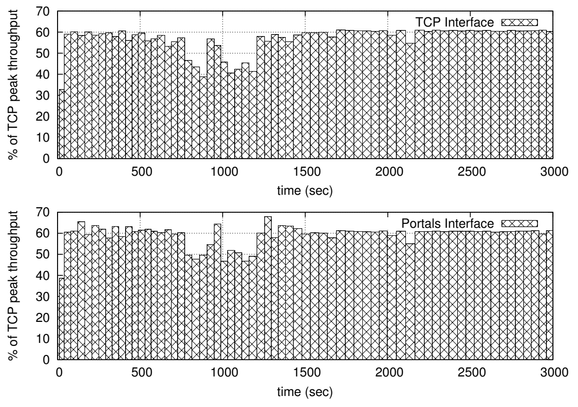


(b) Cumulative I/O time at the streaming server servicing 2048 compute nodes.

Figure 9: Cumulative I/O time times for the GTC simulation measured at the DART streaming server.



(a) DART streaming server servicing 1024 compute nodes.



(b) DART streaming server servicing 2048 compute nodes.

Figure 10: Throughput at the DART server using Portals (compute nodes to service node) and TCP (service node to remote node).

utilized and becomes the bottleneck. From Fig. 9(a) we see that TCP transport is slower than Portals, but as the two transports share the same communication buffer at the streaming server, the Portals throughput adapts to the slower link.

To summarize, in the experiment on 1024 compute nodes, the simulation ran for 1500 sec and each node generated 550MB of data, resulting in a total of 555GB across the system. The overhead on the simulation was less then 0.7%. *In the experiment on 2048 compute nodes, the simulation ran for 3000 sec, with each node generating 550MB of data for a total of 1.2TB across the system and the overhead on the simulation of* 0.4%.

Integration with XGC-1: We have also integrated DART with the XGC-1 [3] simulation. XGC-1 is a edge turbulence particle-in-cell code, that uses numerical models to simulate a 2D dimensional density, the temperature, the bootstrap current and the viscosity profiles in accordance with neoclassical turbulent transport.

In our experiments with XGC-1, we used 1024 compute nodes and one instance of the streaming server. The server allocates the same set of resources as in the GTC case. We set up the XGC-1 simulation to run for 100 iterations, and to produce a checkpoint every 20 iterations. One application iteration takes 30 sec to complete, and a checkpoint is 76 MB in size for a simulation with 1 million particles. The behavior and performance of DART observed in these experiments was similar to that described above for GTC. The XGC-1 simulation ran for 3020 *sec*, and each node generated 760MB of data, resulting in a total of 380GB of data across the system. The overheads on the simulation due to DART operations was $< 0.3\%$ of the simulation compute time. An average sustained throughput of 60% of the peak TCP throughput was achieved between streaming server and remote node.

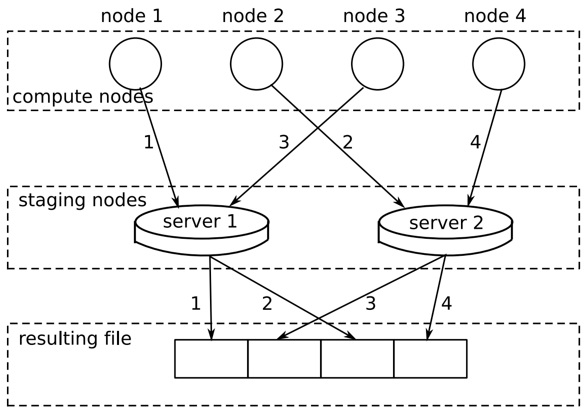


Figure 11: Staging nodes execution flow

4.4 Staging Nodes

In the experiments presented so far, we used a single instance of the DARTSServer, which streamed the data to a DARTReceiver instance that was running on a node on a remote cluster. The DARTReceiver instance received the data and then dumped it. In this experiment we use multiple instances of the DARTSServer and we run them in cooperative mode to save data from the application to a local storage system for later use, e.g., for the application restart. As we use multiple server instances, we also increase the number of compute nodes that we use to run the application on. In this experiment we analyze the scalability of DART on a larger number of compute nodes, i.e., the server component behavior as well as the client layer, and the overhead that DART imposes on the running simulation.

For the setup of this experiment we use again the GTC scientific simulation, and we run it on 8192 compute nodes. The GTC application compute the numerical models for the plasma turbulence, and at regular timesteps streams its new state, i.e., information about particles, through DART. We tuned the input parameters for the simulation to resemble the behavior of a production run, i.e., to produce a check-point file of **XMB** every **X** simulation timesteps. Specifically, we set the *micell* to **X**, *mecell* to **X**, *npartdom* to **X**, *mzetamax* to **X**, and *nphi* to **X**.

On the server side we use 8 DARTSServer instances that service the notification requests from the compute node, extract the data and save it to local storage. The number of dedicated service nodes that are publicly available is reduced. Moreover, they are shared between various user jobs, e.g., application compiling, data archiving, and can impose heavy loads on the system, and thus can impact negatively the performance of the streaming server. To alleviate these problems, we use *staging nodes* to run the instances of the DARTSServer. The staging nodes are a sub-partition of the application compute nodes, that we use exclusively for running the server. We can view the DARTSServer instances as a smaller task in the system, in addition to the GTC simulation.

We present the data flow overview for this set-up in Fig. 11. Multiple compute nodes of the application can write to the same file, yet they may be associated with a different server instance. To keep the content of the file coherent, the DARTClient layer on the compute nodes coordinate among the nodes and determine the proper offset that each node should write at. The numbers at the top of the figure represents a possible sequence of notification posts to the streaming server, and the numbers at the bottom of the figure represents a possible sequence of data extraction and save to the local store. The two sequences are different, but the resulting file is coherent with the compute nodes as a result of the coordination at the DARTClient layer. The DARTSServer instances do not have to coordinate between themselves while writing the to resulting file, because each notification message contains a unique offset for the data block of the corresponding compute node. Implementing the offset coordination function in the DARTClient layer has several advantages over the streaming server, and these include (1) the application on the compute nodes is a SPMD program that executes in incremental time steps and all the processes will reach the streaming stage at about the same time, (2) notification messages are self-contained and if a data transfer has an error, they allow both the application and the streaming servers to continue execution, and (3) the streaming servers can schedule and extract data from compute nodes independently and can achieve a better throughput.

When a user runs a job in the system, he first submits a script describing the job to the queuing system, and then the queuing system schedules and runs the job. Depending on the configuration of the scheduling policy and the number of requests, the system may run smaller jobs more frequently than larger ones. This impose an extra challenge to our staging nodes approach, because we need to synchronize the two jobs to execute at the same time. Our solution to this problem is to combine the two jobs, i.e., the DARTSServer instances and the GTC application, in a single job script that contains two sub-jobs. In the job script file, we first run the streaming server sub-job and put the process in the background, and then we run the application sub-job, and at the end of the job execution we wait for both of the sub-jobs to finish.

present and discuss the results ...

5 Related Work

Related research efforts addressing data I/O issues for large scale parallel applications include parallel file systems such as Lustre [6] and GPFS [8], parallel file I/O infrastructures such as MPI-IO and ROMIO [15], and asynchronous data I/O and streaming infrastructures such as PDIO [13], LIVE/PBIO [4] and DART. Since PDIO and PBIO and most closely related to DART, they are discussed in more detail in this section.

The LIVE Data Workspace project [4] at Georgia Tech and the underlying PBIO layer, has similar overall goals as DART. However, there are several difference in the design and the two systems make different tradeoffs. The focus of PBIO is on reconfigurability and the ad hoc connections of components without a priori knowledge of data requirements. PBIO also supports multiple transports including Portals and Inifinband. DART, in contrast, is more lightweight and is specifically tuned for low latency low overhead streaming, and was is found to perform faster in the tests conducted at the Oak Ridge National Labs.

PDIO [13] is a related effort, and is specially designed to support runtime visualization of data from parallel simulations. It tries to virtually extend the file-system of Portals-enabled compute nodes to arbitrary destination locations. Unlike DART, the design of PDIO is tightly coupled to the requirements of its target applications.

Asynchronous I/O API (ADIOS API) is a recent effort that aims to provide a simple I/O interface capable of integrating with a variety of data transport layers, both synchronous and asynchronous. The goal of ADIOS is to provide an interface nearly as simple as standard Fortran I/O statements, while giving access to the power and flexibility of mature systems such as MPI-I/O, HDF-5 [5], parallel netCDF [12], standard POSIX calls, as well as more experimental systems such as DART and the Georgia Tech LIVE system. Additionally, the data is encoded in a tagged, binary format with data and grouping attributes for easier data use without the overhead of a plain text data format. The system consists of three parts: (1) the API that was designed primarily for Fortan use, but also tested with C, (2) an XML configuration file describing the data types, attributes, data transports, and buffer specifications selection, and (3) the data transport layer, e.g., DART.

The idea of asynchronous calls by overlapping computation with I/O data transfers was explored in other projects as well, e.g., Kangaroo [14]. However, the focus of the Kangaroo project was to improve application reliability, while the overhead on the application was not a concern. Moreover, Kangaroo and other similar projects implement this idea in a multithreaded environment where the application thread can make progress while the streaming thread blocks. Our system can run in single-threaded environments, e.g., HPC resources tuned for scientific computing, and a data transfer does not block the running application.

The DAFS [11] project is a file system implementation in user space, that provide access to remote file systems using RDMA data transfers. DAFS has a similar approach of providing asynchronous calls to the application layer, e.g., it provide the abstraction of a mounted file system which is accessed with RDMA calls. Our approach is simpler, as it does not implement the complete file API, and provides several advantages because of this (1) it has a reduction factor in the number of clients accessing the file system at the same time, and this reduces the concurrency on the metadata server, and further (2) the streaming servers can cache and share a file descriptor, and thus a connection to remote file for multiple compute nodes.

6 Conclusions

In this paper we demonstrated that the DART data transfer substrate can be effectively used to address the growing data transport requirements of large scale parallel simulations on current HPC systems. DART is built on the RDMA technology, and enables fast, low-overhead and asynchronous access to data from a running simulation, and supports high-throughput, low-latency data transfers. **DART provides an asynchronous transfer API for single threaded environments that are common on HPC resources tuned for scientific computing.** We presented the design of DART as well as its implementation on the Cray XT3/XT4 system using the Portals RDMA framework, and an evaluation using the GTC and XGC-1 simulations from the Center for Plasma Edge Simulation FSP. The evaluation demonstrated that DART can saturate the SeaStar link between compute and service nodes with 4MB or larger messages. The results also demonstrated that the overhead of DART on the application was a very small percentage of the computation time. Experiments with DART integrated with the GTC simulation also demonstrated scalability, low overheads, low latency and high throughput.

Current and future research efforts include experimenting with other application codes, building an end-to-end code-coupling layer on top of DART, and porting DART to other RDMA platforms such as the BlueGene/P and Infiniband.

References

[1] `http://info.nccs.gov/resources/jaguar`.

[2] Ron Brightwell, Trammell Hudson, Kevin Pedretti, Rolf Riesen, and Keith Underwood. Implementation and Performance of Portals 3.3 on the Cray XT3. *IEEE International Conference on Cluster Computing*, September 2005.

[3] C.S. Chang, S. Ku, and H. Weitzner. Numerical study of neoclassical plasma pedestal in a tokamak geometry. volume 11, pages 2649–2667, 2004.

[4] Karsten Schwan Hasan Abbasi, Matthew Wolf. Live data workspace: A flexible, dynamic and extensible platform for petascale applications. In *Cluster Computing*, Austin, TX, September 2007. IEEE International.

[5] HDF-5. http://hdf.ncsa.uiuc.edu/products/hdf5/index.html.

[6] Cluster File Systems Inc. Lustre: A Scalable, High Performance File System. `http://lustre.org/docs/whitepaper.pdf`.

[7] Cray Inc. Cray XT3tm Sytem Overview. Technical Report S-2423-13, November 2005.

[8] IBM Inc. Gpfs: A Shared-Disk File System for Large Computing Clusters. `www.almaden.ibm.com/StorageSystems/projects/gpfs/Fast02.pdf`.

[9] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid-Based Parallel Data Streaming implemented for the Gyrokinetic Toroidal Code. November 2003.

[10] Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, and R. B. White. Turbulent transport reduction by zonal flows: Massive parallel simulations. *Science*, 281(5384):1835–1837, 1998.

[11] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo I. Seltzer, Jeffrey S. Chase, Andrew J. Gallatin, Richard Kisley, Rajiv G. Wickremesinghe, and Eran Gabber. Structure and Performance of the Direct Access File System. In *Proc. of USENIX Annual Technical Conference*, June 2002.

[12] Parallel netCDF. http://trac.mcs.anl.gov/projects/parallel-netcdf.

[13] Nathan Stone, Doug Balog, Bryon Gill, Brian Johanson, Jim Marsteller, Paul Nowoczynski, David Porter, Raghurama Reddy, J. Ray Scott, Derek Simmel, Jason Sommerfield, Katie Vargo, and Chad Vizino. Pdio: High-performance remote file i/o for portals-enabled compute nodes. In Hamid R. Arabnia, editor, *PDPTA*, pages 925–930. CSREA Press, 2006.

[14] D. Thain, J. Basney, Se-Chang Son, and M. Livny. The Kangaroo approach to data movement on the Grid. In *Proc. of 10th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 325–333, August 2004.

[15] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1999.