Grid -Based Parallel Data Streaming implemented for the Gyrokinetic Toroidal Code

S. Klasky[1], S. Ethier[1], Z. Lin[2], K. Martins[1], D. McCune[1], R. Samtaney[1]

[1]Plasma Physics Laboratory, Princeton University, NJ 08543 USA

{sklasky, ethier, kmartins, dmccune, samtaney}@pppl.gov

[2]Department of Physics and Astronomy, University of California, Irvine , Irvine, CA 92697 zhihongl@uci.edu

Abstract

We have developed a threaded parallel data streaming approach using Globus to transfer multi-terabyte simulation data from a remote supercomputer to the scientist's home 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 then transferring this data to be post-processed. The present approach is conducive to using the grid to pipeline the simulation with post-processing and visualization. We have applied this method to the Gyrokinetic Toroidal Code (GTC), a 3-dimensional particle-in-cell code used to study micro-turbulence in magnetic confinement fusion from first principles plasma theory.

1. Introduction

A major issue preventing fusion devices from achieving ignition has been loss of confinement due to cross-field transport. Energy transport from the hot and dense core of the plasma to the cold walls of the device greatly exceeded the level predicted by the earlier theory of Coulomb collisions. It is now believed that plasma microturbulence driven by temperature and density gradients are responsible for these enhanced cross-field transport rates. The ability to suppress microturbulence-driven transport may well be the key to a practical magnetic confinement fusion device. The Gyrokinetic Toroidal Code (GTC) was built to develop the necessary foundation of physics understanding. These highly complex, nonlinear phenomena of plasma turbulence can be most effectively investigated using direct numerical simulations. We have developed a particle-in-cell (PIC) turbulence code to study the important wave-particle interactions in high temperature plasmas. With recent advances in physics models, numerical algorithms, and power and capacity of massively parallel computers, we were able to carry out whole-device simulations of plasma turbulence with unprecedented realism and resolution. These advanced simulations have reproduced key features of turbulent transport observed in fusion experiments[1], and have stimulated further theoretical and experimental research in the world fusion community.

In this paper, we describe a technique for achieving efficient data transfer from a supercomputing application (GTC described below) to a local cluster for analysis and visualization *using a pipeline approach implemented using Globus and pthreads.* Section 2 describes the GTC code along with some of its computational techniques. Section 3, describes the computational performance of GTC. In Section 4, we briefly describe the visualization techniques used in the analysis of GTC data. Section 5 describes three basic data management methodologies used by fusion researchers. Section 6 describes our "grid" comprised of Linux clusters at PPPL and Princeton University (PU), followed by the implementation of the threaded streaming techniques. In Section 7, we present the results obtained from data transfer experiments on the PPPL-PU grid. It will be shown that the entire process of large-scale simulations and simultaneous data transfer to local clusters for visualization/analysis leads to the best overall performance/productivity. Finally, we conclude with future work in Section 8.

2. The Gyrokinetic Toroidal Code (GTC)

Compared with hydrodynamic turbulence, plasma microturbulence exhibits more complexity due to the interaction of the charged particles with a combination of externally imposed and self-generated electro-magnetic fields. We have developed a time-dependent 3-dimensional particle-in-cell (PIC) code that solves the gyrokinetic system of equations for particle motion in a plasma [2,3]. In a magnetic confinement device, the trajectory of a charged particle (ion, electron) is a fast helical motion along the magnetic field lines accompanied by a slow drift motion across these same lines. It was found that by averaging the fast helical motion, huge savings in serial computational time were achieved without losing any of the relevant physics at the length and time scales of the problem under scrutiny. A charged ring moving along and across the field

SC'03, November 15-21, 2003, Phoenix, Arizona, USA

lines thus replaces the helical motion of the particle. This is known as "gyrophase averaging", and we end up following only the (guiding) center of the ring instead of the full motion of the charged particle. The gyrokinetic system of equations is obtained by gyrophase averaging the Vlasov and Poisson equations in the electrostatic case [2,3]. One of the innovations in our code is the use of a local method, known as the four-point averaging, instead of a spectral Fourier method to solve the Poisson equation. This greatly reduces the computational work and is more easily parallelized [3,4].

2.1 Particle-in-Cell method

The PIC method consists of using "particles" to sample the distribution function of a system in phase space and follow its evolution in time. Interactions between the particles are handled using a self-consistent field described on a grid.

The following are the main steps of the PIC algorithm. First we deposit (distribute) the charge of each particle on its nearest grid points (this is the "scatter" operation). Then we solve the Poisson equation on the grid to find the electrostatic potential and field at each grid point. We then calculate the force acting on each particle from the field at its nearest grid points (this is the "gather" operation). Finally, we move the particles according to the forces just calculated, and repeat these steps until the end of the simulation.

Since the original PIC algorithm can be fairly noisy in certain cases, GTC uses the delta-f method, which consists of following only the non-equilibrium part of the distribution function instead of the full function [5]. The equilibrium is part of the "background" in the equations and does not change during the simulation. This greatly reduces the numerical noise and allows a near-uniform distribution (in space) of the particles even in the presence of density gradients.

2.2 The GTC Mesh

Because of the characteristic motion of the ions in the externally applied magnetic field, moving fast along (parallel to) the field lines and slowly across (perpendicular to) them, a magnetic coordinate system is the natural choice in gyrokinetic calculations. Consequently, typical eddies in plasma turbulence are elongated along magnetic field lines. Using a mesh aligned with the magnetic field, a factor of 100 in serial computation time can be saved. This is due to the fact the trajectory of a charged particle along the field line is aligned with one of the coordinate axes. Furthermore, aligning the mesh with the field lines allows for a much larger time step and a much smaller number of grid points in the parallel direction (i.e., along the field lines). Figure 1 shows the toroidal mesh (long way around the torus) used by GTC. We see that the field lines twist around the torus and so does the mesh. Figure 2 shows a perpendicular section of the torus, or poloidal plane. To maintain the density of grid points constant for different radii, the radial and angular meshes have regular spacing, i.e. Δr_i = constant and $r_i \Delta\theta_i$ = constant. This makes the grid irregular since the number of points per radial surface (also called flux surface) varies from one surface to the other, and the points do not align radially.

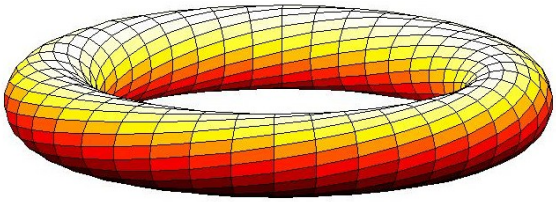

Figure 1. Representation of the field-line following mesh on a flux (magnetic) surface of the system (constant radius). The twist in the field lines depends on the magnetic equilibrium of the device under study.

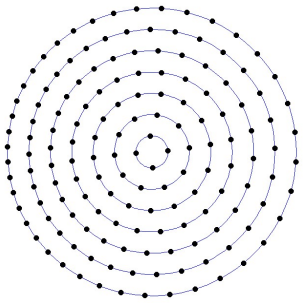

Figure 2. Mesh of a poloidal plane (perpendicular section) showing the constant density of points. This mesh rotates as one goes around the torus due to the twisting of the magnetic field lines.

2.3 Parallel approach

There are two parallel methods implemented in GTC. Coarse-grained one-dimensional domain decomposition splits the torus in equal partitions containing the same number of grid points (poloidal planes) and, initially, the same number of particles. MPI calls take care of the communication between processes. A second level of parallelism in the code uses fine-grain loop-level work splitting using OpenMP compiler directives. This mixed-mode MPI/OpenMP approach is ideal for clusters of shared memory nodes, like the IBM SP, on which most of the GTC simulations are performed. It is well known that loop-level parallelism does not scale as well as coarse-grain domain decomposition. However, the physics of our system dictates the number of "useful" domains that can be used for the 1D decomposition. Beyond a certain grid resolution in the toroidal direction, no changes in the simulation results occur, and it is then unnecessary (and wasteful) to increase that resolution any further in order to use more

domains (we can actually use as little as one poloidal plane per domain). This unusual property is due to our global field-aligned mesh and to the plasma physics phenomenon known as Landau damping. Loop-level parallelism with OpenMP allows us to put more processing resources towards our large simulations.

Future work will include adding an extra level of domain decomposition in order to "recruit" even more processors for even larger and more demanding simulations. Currently, large simulations use 1 billion particles and 125 million grid points.

3. GTC performance

GTC has been ported to most parallel computers. The production simulations are mostly done on the 10 teraflop IBM SP (Power3) at the National Energy Research Scientific Computing Center (NERSC). The large runs use 64 MPI processes (1 per SMP node) with 16 OpenMP threads per process, for a total of 1,024 processors. Message passing communications account for only 5 to 8% of the total wall clock time on the IBM SP Colony switch. Parallel efficiency of the domain decomposition part is essentially perfect for all ranges of problem sizes, in contrast with the fine-grain OpenMP, which depends on the problem size but can reach 98% of efficiency for sufficiently large cases. Figure 3 shows the scaling of GTC on 2 different platforms: the CRAY T3E and the 16-way SMP IBM SP Power3 at NERSC. The metric that we use is the computing power, defined as the number of particles, in millions, which move one step in one second (wall clock). We believe this to be a much better quantity to measure the efficiency of a PIC code than the floating-point operations count or just pure parallel scaling.

Half a million particles per processors were used to get the IBM SP scaling curve in Figure 3. The somewhat sudden change in the curve past the 64-processor mark is due to the mixed-mode MPI/OpenMP. The loop-level parallelism does not scale as well as the coarse-grain domain decomposition but would do better as the problem size is increased (more particles, higher grid resolution).

The MPI scaling of the domain decomposition works well for all the platforms that we have run on, including Linux clusters with fast Ethernet (100 Mbs). This is due to the efficient message passing algorithm implemented in the code. The bulk of the communications happens when some particles move out of one domain to enter another. This is done in a closed chain fashion, each processor receiving from the left while sending to the right, and then vice versa. This method optimizes the communications by avoiding the case when one processor would receive from two others or more.

For medium-size simulations and a relatively small number of processors (32 CPUS), we have found that GTC runs as fast on the latest 32-bit AMD and Intel processors as on the IBM Power3. This was observed on Linux clusters equipped with a myrinet interconnect in one case, and a gigabit-Ethernet interconnect in the other.

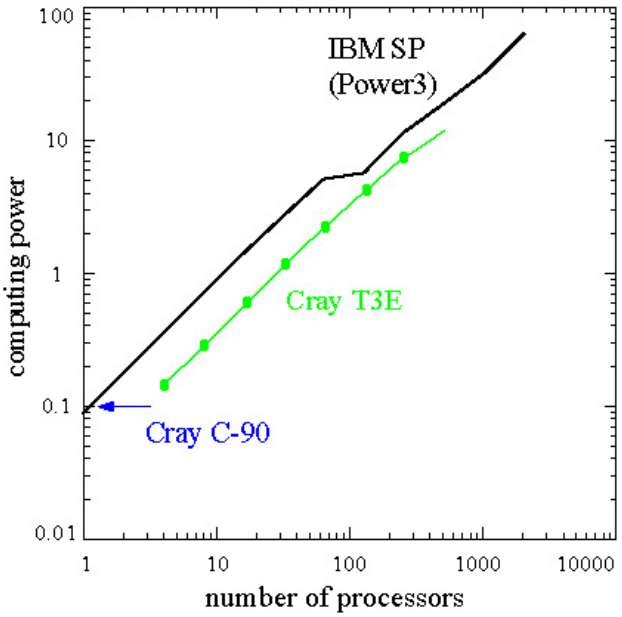

Figure 3. Scaling of the computing power of the GTC code as the number of processors is increased. This quantity is defined as the number of particles (in millions) that move 1 time step in 1 second. The change in the scaling which appears at >64 processors is due to the OpenMP.

4. Visualization & Data Analysis

The GTC code produces HDF5 files of the electrostatic potential. After we generate these files, we run AVS/Express] to visualize the results. The visualization is working off of a node on our local cluster, and can process frames at about 15seconds/frame for some of our larger runs (See Figure 10). An ongoing activity at PPPL is "run-time monitoring" of simulations. To achieve this, we have implemented socket connections inside of Express, in order to receive data from the I/O servers in real time, and write out the image frames. Furthermore, there is an ongoing effort to work with parallel visualization programs (e.g. Ensight Gold).

In the future, we would like to implement an out-of-core isosurface routine, and then put this into our visualization software. Once this is done, a user can later generate isosurfaces at a much faster rate than what is currently being done. This part of the analysis will be done as soon as the data comes over to our servers, which is a natural extension to the work presented in this paper.

Parallel analysis/visualization will become imperative as the GTC runs get larger. We think it is advantageous to simultaneously stream data in parallel from N processors on the SP to M processors on a local PPPL cluster in order to maintain processor affinity which can be exploited by parallel analysis/visualization routines. This is the underlying motivation for the work described in later sections.

5. Data management methodologies

Presently, fusion HPC codes generate a vast amount of data, and researchers transfer this data from a supercomputer to their local analysis/visualization resources after the simulation has completed. In general, most analysis/visualization tools are serial in nature. Performance limitations render this method of working impractical for very large datasets.

Grid computing has made a major impact in the field of collaboration and resource sharing [6]. Data Grids are generating a tremendous amount of excitement in grid computing [7]. There is an ongoing effort by visualization scientists to use the grid to develop collaborative interactive visualization techniques [8, 9, 10, 11, 12].

Our approach for codes such as GTC is to transfer data from a supercomputer running the main simulation on N processors, to M processors (typically M<N) on a visualization/analysis cluster local to the scientist. Typically, post-processing of the data requires much less computing power than the main simulation. By maintaining the distribution of the data among processors, we can then post-process this data in parallel on the local resources with parallel analysis/visualization techniques. We realize that different post processing techniques might require a different distribution of data than that used in the original calculation. At present, we let the application scientist decide the distribution on the visualization/analysis cluster, but in the future we would like to automatically redistribute the data according to the characteristics of the visualization/analysis cluster. Before we begin the description of our approach, we would like to survey other methods used in the fusion community for analysis and visualization of supercomputing data.

In the magnetic confinement fusion community, three methods are commonly used to analyze data from a supercomputing simulation. The first method uses a database system known as MDSPlus [13]. The second approach is to write data to a local disk on the supercomputer. To simplify the process of assembling files from all of the processors, it is much more convenient to write the data in parallel, from all processors to a single file. This file may be on a network-mounted disk (such as NFS), or on a parallel file system (such as PVFS or GPFS). We rely on MPI-IO for this process. The third approach, which is emphasized in this paper, is to thread the I/O layer and use Globus/GridFTP to stream the data from the supercomputer to a local cluster for data analysis. In the next few paragraphs, we will briefly describe these three methods.

MDSPlus is a client-server system for the acquisition, analysis, storage and sharing of data. When users want to examine a variable after their simulation has run, they can request that variable, or even a reduced-dimensional slice from this variable, and just have this data sent back to him/her. There is a significant disadvantage of MDSPlus for large datasets. The data from a HPC code is sent to a serial server, using serial socket connections from one processor. Such serialization of I/O is unacceptable for HPC codes, except perhaps for a few small summary "results" datasets. Because of this problem, and others, we did not implement MDSPlus into GTC.

The second approach is to write data to a local disk on the supercomputer and later transfer the data using protocols such as ftp over to the local analysis/visualization cluster. Presently, in this approach the scientist either: (1) writes files to an NFS mounted disk; (2) writes separate files on each processor, and later assembles these into a single dataset; (3) writes hdf5 files on a parallel file system. Since these three methods are commonplace in the fusion community, we compare these three methods with our preferred approach based on streaming. It is important to note that most researchers in the fusion community who run on clusters write to a NFS mounted disk. In GTC we chose to write the files in the HDF5 format because it contains meta-data, it is binary portable across most architectures, and most importantly, it allows efficient parallel I/O on a parallel file system [14]. The third approach of parallel data streaming is described in the following section.

6. Threaded Parallel Data Streaming

Interactive remote data analysis/visualization is inconvenient at remote supercomputing centers like NERSC where generally jobs are executed in a batch fashion. Even with the availability of interactive resources at remote sites, issues such as latency and network quality of service, hamper productivity. Clearly, it is advantageous for GTC scientists to move the visualization and data analysis to a local resource. To achieve this objective, we have implemented a secure threaded parallel data streaming method described below.

6.1. Overview

Analysis and visualization of GTC results are an important part of understanding the physics of micro-turbulence. Since the total data size from a GTC run can be quite large, of the order of 1 or more TB of data, analysis and visualization routines often need to be run as multiprocessor applications in order to process the data in a reasonable amount of time. Interactive visualization becomes impractical because reading and displaying the data take several minutes for every frame. Speed of access plays a major role in determining the use that is ultimately

made of the data. In order to ask "what-if" questions, the analysis/visualization routines and their users require high performance access to the data.

Although analysis and visualization may require some parallel processing, it does not require the 1000s of processors needed by the original simulation. Storage and processing power have become much less expensive for clusters, and networks have become faster: this is one of the basic tenets of grid computing. Hence, we find that the analysis and visualization of simulation results is not only feasible but desirable to take place on a local machine that is physically separated from the supercomputer running the simulation. This permits the great performance advantage of placing the results data directly on hardware and networks local to the scientist investigating GTC results. In the past few years, the time to transfer large datasets from the supercomputer center to local systems was an impediment. Now, however, with our current implementation of threaded streaming data transfer based on GridFTP, this transfer can occur while the GTC simulation is still running. This mode of data transfer incurs a small overhead and does not affect the performance of GTC on the supercomputer by any significant measure (See Section 7 on results from data transfer experiments).

As a first step in developing the above mentioned data transfer scheme, we created a test bed of 3 clusters separated by a WAN. This test bed serves as a controlled platform for performing data transfer experiments, investigating various scenarios and testing of our APIs. Figure 4 is a schematic of our network topology from two separate clusters/supercomputers to our local cluster. The cluster on the top is an AMD dual processor (MP2100) cluster with 9 compute nodes, and one head node connected to a 3Com gigabit switch. This cluster is almost identical to the cluster at PPPL on the bottom(the only difference is that PPPL's cluster has 18 compute nodes, with one head node, and one visualization node). These two clusters are separated by a 100Mbit microwave link, and a few routers. We have just finished our experiments on the PU – PPPL clusters. We have just started the next phase of data transfer experiments between the machines at NERSC, which is on the ESNET, and the local PPPL cluster. Currently, our connection to the ESNET is limited to 155Mbit (OC3).

6.2. Threaded data transfer methodology

In this Section, we give an overview of the basic transfer scheme which we have implemented. We have built several APIs in C, with FORTRAN90 hooks, to perform our goal of streaming data from a live simulation on a remote supercomputer to a cluster which is local to the user.

Our system is built on top of the Globus toolkit, namely the APIs which are used for GridFTP [6,7]. In our system we use POSIX pthreads to thread the I/O layer. Data is copied from the main program to a buffer. The thread is then activated and starts to stream a small piece of this data using GridFTP APIs. Data is then streamed from the supercomputer to our local cluster on which each node runs a GridFTP server. Typically, we send ten GridFTP streams per processor. Data is written to disk on the nodes of the local cluster. If all of the data from a variable has been successfully transferred, the server initiates a MPI program to create a HDF5 file. If pieces of the data are missing, the user runs a separate program at the end of the job to get the missing data, and convert this raw-data into HDF5 Files.

We run a slightly modified GridFTP server on each node on our local cluster in order to extend the data pipeline approach. In addition to writing the data to local disk, the GridFTP service include APIs to communicate with a local master node which can coordinate such tasks as assembling the local files into one parallel HDF5 file, perform analysis routines, etc. These servers will accept the incoming data, process the data, and then write this processed data to disk.

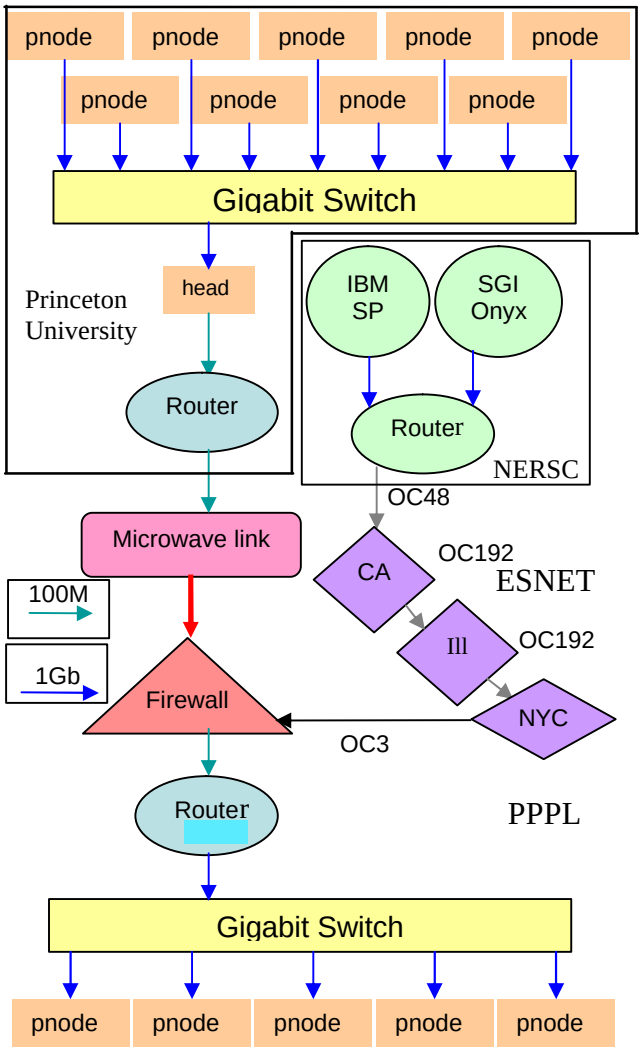

Figure 4 There are three target architectures for this paper; Princeton University, PU (top), NERSC (middle right), and our local cluster, bottom. The first architecture is a cluster of 9 dual processor AMD 2100MP nodes with a gigabit infrastructure. The second is the IBM SP and SGI at NERSC. The third is a cluster local to PPPL with 18 dual processor nodes; where we only show 5 nodes. PPPL has an OC3 (155Mbit) connection to the ESNET, and a 100Mbit connection to the cluster at PU.

6.2.1. HDF5-part. It is very important that the data from the GTC code be eventually written to disk. Our scheme will write this information to the disk on the local cluster. Since we intend to write HDF5 files, we need to be able to send out meta-data (labeled as "HDF5_info" in Figure 5) along with the raw data from GTC so that we can reformat the data file into a HDF5 file on our local cluster. The metadata necessary to send over to create a HDF5 is approximately 2K, which is several orders of magnitude smaller than the files which we transfer. We plan to include keywords to inform the Globus server which "filter" routines will run after the raw data arrives. These filters will allow us to locally post process the data on-the-fly.

6.2.2. Thread/Buffering Part.

Our threading approach is conceptually similar to that of Ma et al. [15]. During the simulation, instead of writing local files to the parallel file system, the information is organized and copied into the memory buffer (see Figure 5). The thread then transfers the data from the buffer. In the event of network disruption, or buffer overflow, the thread writes a binary file to its local disk. In addition to the data, a status file is either written locally or transferred. The status file contains pertinent information about the data transfer of each block in Figure 5. In our current implementation, a clean-up procedure is initiated manually (eventually automated) by the user at the end of the simulation. This clean-up procedure, which uses Globus/GridFTP, examines the status files and transfers the remaining data at the remote supercomputer site to the local cluster.

Figure 6 shows the main streaming routines used by GTC. The user allocates memory for a buffer, which is on each processor that performs the I/O (possibly one per SMP node); to account for the meta-data, the user will create a buffer which is slightly larger than the amount of data he/she wishes to transfer at any one output step of the simulation. When the user first calls the open statement (`globus_topen`) on each processor, it creates a new thread and passes some initial data that informs the thread the location of the memory buffer. As in [15] we only create one thread per processor. These threads do not communicate with the other I/O threads on the other processors. The thread monitors a queue of transfers which is protected by a pthread mutex lock, and a variable which sets up a condition. Later, when the user wants to write data, he/she uses the `globus_twrite` function. This allows the main thread to append the new transfers to the queue. The background thread is then able to continuously transfer information to the GridFTP service on the remote machine. When the background thread has finished transferring all of the information in the buffer, it then sleeps. At the end of the date transfer, the code calls `globus_tclose`. This allows the background thread to write any information in the buffer onto disk, and then rejoin the main thread. A call to `globus_tclose` precedes the call to `mpi_finalize`.

The thread manager keeps track of specific values which relate to the buffer, such as the location of the new data, the location of the next transfer point, and the length of the buffer. This is shown in Figure 6. The thread manager values must be updated whenever the data is added or removed from the buffer, and it is equipped to wrap around from the end of the buffer back to the beginning. Thus we use a simple memory management scheme, keeping track of the memory via a simple linked-list. If new data will not fit into the free space in the buffer, we guarantee that the buffer does not overflow by writing this new data to disk. After the buffer sends the data to the server, it frees that portion of memory.

Instead of sending each buffered dataset in one big chunk, we break the data into manageable blocks and send each block separately. In the event of a network interruption, a block may not be transferred. In this case, the system senses failure and writes this block to the local disk. After the failure, the code continues to attempt to transfer the next block of data. It is necessary to ensure that the data transfer be lossless. Therefore, we only keep two states, success or failure. If we have a success in our transfer, we free up the memory in the buffer, and continue to transfer. If we have a failure, then we write the block of data to disk, and then free up the memory in the buffer. In both cases, we write a status file so that we can keep track of the location of the data. We also stream this status file over to our local cluster, but if this fails, we write this to the local disk. Status files are used at the end of the simulation, to get the missing blocks.

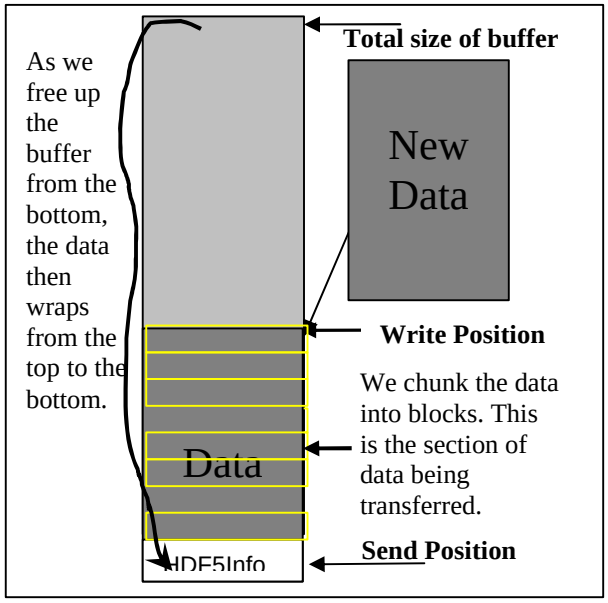

Figure 5. The data is broken up into blocks, which is shown in bottom of the figure. These blocks indicate the size of the data to transfer for each stream on each processor. Typically, we use ten streams per processor.

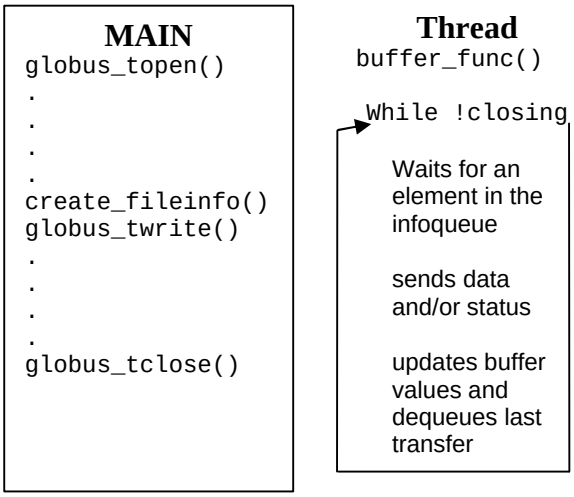

Figure 6. Routines the simulation code calls, along with the thread-mechanism. Globus_topen initiates the thread. Globus_twrite copies the data into the memory buffer, then adds the transfer information into the queue, and then updates the buffer values. Create_fileinfo writes metadata to the buffer. Globus_tclose sends the close signal to the thread, and then copies the remaining data to disk. The buffer_func waits for the transfer information in the queue. It sends data to remote machines in blocks followed by buffer updates, finally removing the completed transfer information from the queue.

7. Results

One of the main objectives of our approach was NOT to slow down the GTC calculation on the supercomputer. Therefore we compare the time of the GTC code using our streaming routines to running the code with only writing to local disks. In this Section, we compare the time it takes to run the code with (a) no I/O, (b) writing binary data, using FORTRAN write statements to disk, (c) writing an HDF5 file to a parallel file system, (d) writing the HDF5 file to a NFS-mounted file system, and (e) streaming the data. See Table 1. Each quantitative result presented in this Section is an average of 50 runs of GTC. The percentages in columns (b-e) are the %-overhead of the code with I/O compared with no I/O whatsoever. The LOCAL case is where we use FORTRAN write statements on a local disk. Obviously, this is the fastest method to write output, but unfortunately, it is not binary portable across platforms, and will have to be processed on the supercomputer before it can be transferred. Streaming data with a threaded I/O layer produces the next smallest timings in all of the runs on the AMD cluster, making the code run on average 10% slower. On average, this is better than even the runs on the SP writing to the GPFS directory, where the typical slowdown ranged from 24% for the larger runs. Since we are using 8 nodes out of 10 nodes on the cluster for these runs, the CPU load should be 80%, which is true for cases (a), (b), (c), and (e). Since our target architectures always have either a head node (as in the PU cluster), or an I/O node (as on the IBM SP), we didn't compute on all 10 nodes

Some of the overhead from the streaming routines comes from the memory copy from the array in the main thread to the buffer. This is very similar to the overhead when writing local I/O with a FORTRAN `write` statement. The rest of the overhead is mostly due to the overhead of the thread routines trying to transfer the information. For Run 1 the data production rate was faster than the streaming rate resulting in some blocks being written to files on the PU disks.

It is clear that writing data to an NFS mounted disk gives the worst performance in our experiments, and this strategy, which is commonly used on clusters, should be avoided if possible. In Figure 7 we plot the CPU load on the PU cluster for runs writing to NFS (until time 14:50 in the graph). The CPU usage is intermittently dropping drastically to 50% when the NFS writes occur. This is very different compared to writing to a disk on a parallel file system such as PVFS. In Figure 7, for times when we write to PVFS (time after 14:50 on the graph) the CPU load stays somewhat constant, never going down in a dramatic fashion as for the NFS write case. Figure 8 clearly shows that the CPU load on the cluster is 80% when we stream data. A detailed examination reveals a small overhead, up to 10%, due to streaming. This overhead is due to the memory copying (about 1% of CPU utilization, and the GridFTP APIs account for the rest of the overhead).

The curve in Figure 9a shows the network usage on the 100 Mb microwave link for Run 3 with streaming enabled. In this case we averaged about 55Mb, which is very close to the estimate of 49Mbs; obtained from calculating the GB/s produced by the GTC code. Since we send some additional metadata, we use a bit more of the bandwidth in the beginning. Figure 9b shows the network usage across our OC3 ESNET for run 10 with streaming enabled. This run clearly shows that we can stream data from NERSC to PPPL to at least 88Mbs.

Similar experiments, not reported in detail here, with another code, achieved a maximum transfer rate of 93Mbits on our PU PPPL grid cluster. The GTC code cannot produce data for realistic runs at this rate on machines with less than 1024 processors. More realistic runs on 16 processors produce data at the rate of 10Mbs. One of our future goals is to stream data across from one of our largest simulations at NERSC, Run9 shown in Table 2. Typical runs are about 4,000 times steps, which suggest that Run 9 will produce 903GB of data per variable. In order to determine if we could realistically stream data from the GTC code at NERSC to our local cluster, we evaluated the performance of the GTC code, shown in Table 1. We conjecture that data from production GTC runs can be effectively transferred to PPPL using our streaming mechanism when the firewall at PPPL is upgraded to gigabit speed.

8. Conclusions and Future Work

In this paper, we proposed a method to thread and buffer the I/O layer for background processing. We performed parallel data streaming experiments between the PU cluster and local PPPL cluster. We achieved results which showed that we can use 95% of the bandwidth, and by threading the I/O layer, we achieve times which are faster than writing to the local parallel files system. Experiments at NERSC indicate that production runs of GTC generate data at a rate of 100 Mbs which is still compatible with parallel data streaming.

Our future plan includes: (1) Port our routines to the IBM SP at NERSC, (2) Pipeline the analysis and visualization with the simulation, (3) automate the post-run clean-up procedure, and (4) make the system fault tolerant. We also are trying to experiment with MPICH-G2[16] for finer grain parallelism for some of our analysis routines in our computational pipeline.

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. This work was also supported by the internal PPPL funding.

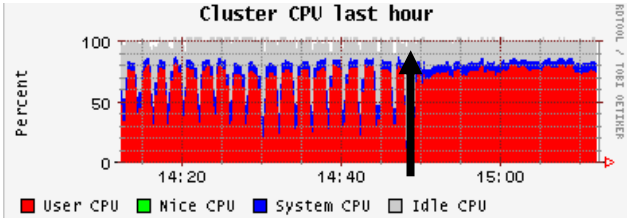

Figure 7. The first portion of this graph (until t=14:50) shows a run where we wrote to an NFS disk. Notice the CPU usage goes down whenever we write to disk. The second portion of the graph (t>14:50) shows a run with writing to PVFS. The arrow demarcates the NFS run from the PVFS run.

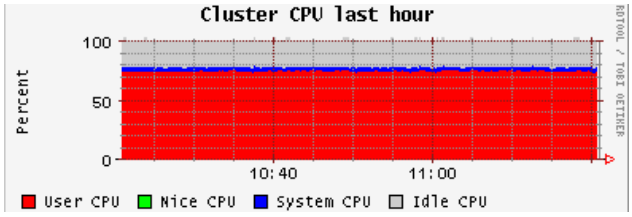

Figure 8. The load on the cluster with streaming .

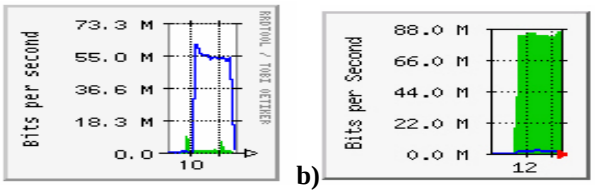

Figure 9. This shows the network usage from Run 3, when streaming data.

Table 1. Comparison of CPU time (in seconds) to stream data versus different methods to write data to local disk. %-overhead is computed by comparing the run with no I/O. All runs were on 16 processors, with 4 variables written per iteration, and 10 time steps. Runs [1-6] produced data at the rate of (167, 75, 49, 41, 30, 19) Mbs. [A=average overhead]. *Run 1 made the streaming run write to disk for part of the data.

id	mesh (particles) (M)	GB data size	(a) no IO secs	(b) local IO secs	(c) pvfs secs	(d) nfs secs	(e) stream (secs)
1	33.6 (4)	5.10	228	238 4%	376 65%	470 106%	250* 10%
2	8.4 (12)	1.32	132	134 2%	176 33%	196 48%	144 9%
3	8.4 (16)	0.13	203	211 4%	249 23%	274 35%	212 4%
4	33.6 (4)	5.10	893	914 2%	1057 18%	1150 29%	1027 15%
5	16.8 (64)	2.50	634	680 7%	713 9%	929 47%	688 9%
6	0.5 (4)	0.08	32	32 0%	46 43%	49 53%	36 13%
A				3%	33%	53%	10%

Table 2. Performance on the IBM SP at NERSC. Run 9 is a realistic high resolution run. Notice that the code was 18% slower because of I/O. Runs produced data at (75, 41, 95) Mbs. Runs 7-8 were on 16 processors, Run 9 was on 1024 processors. [A=average overhead]. All runs were sampled over 10 time steps

id	mesh (particles) (M)	GB data size	(a) no IO secs	(b) local IO secs	(c) GPFS secs
7	8.4 (12)	1.32	228	238 4%	376 65%
8	33.6 (64)	5.10	203	211 4%	249 23%
9	115.6 (512)	2.3	195		241 24%
A				4%	37%

Table 3. Performance on 12 processor SGI Onyx at NERSC using 8 processors, with 10 time steps.

id	mesh (particles) (M)	GB data size	(a) no IO secs	(b) local IO secs	(c) hdf5 local secs	(d) stream secs
10	8.4 (12)	1.28	186	192 3%	N/A	198 6%

References

[1] Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, and R. B. White, *Science* **281**, 1835 (1998).

[2] Lee, W.W., *Gyrokinetic approach in particle simulation*, **Phys. Fluids 26,** 556 (1983).

[3] Lee, W.W., *Gyrokinetic Particle Simulation Model*, **JCP 72,** 243 (1987).

[4] Z. Lin and W. W. Lee, *Method for Solving the Gyrokinetic Poisson Equation in General Geometry*, **Phys. Rev. E 52**, 5646-5652 (1995).

[5] A. Dimits, W. W. Lee, J. Comput. Phys. 107, 309 (1993).

[6] Foster, Kesselman **,** Computational Grids., *Chapter 2 of "The Grid: Blueprint for a New Computing Infrastructure"*, Morgan-Kaufman, 1999.

[7] Allcock, Bester, Bresnahan, Chervenak, Foster, C., Kesselman, Meder, V. Nefedova, D. Quesnal, S. Tuecke. Data Management and Transfer in High Performance Computational Grid Environments. *Parallel Computing Journal*, Vol. 28 (5), May 2002, pp. 749-771.

[8] Bethel, Shalf, *Consuming Network Bandwith vith Visapult*, The Visualization Handbook. Ed. Hansen, Johnson, Academic Press, 2003.

[9] Beyon, Kurc, Sussman, Saltz, *Design of a Framework for Data-Intensive Wide-Area Applications*, Heterogenous Computing Workshop , pp. 116-130. 2000.

[10] Karonis, Papka, Binns, Bresnahan, Insley, Jones, Link, *High-Resolution Remote Rendering of Large Datasets in a Collaborative Environment*, Preprint ANL/MCS-P1030-0203, Feb. 2003.

[11] Beyon, Chang, Catalyurek, Kurc, Sussman, Andrade, Ferreira, Saltz, *Processing Large-Scale Multidimensional Data in Parallel and Distributed Environments*, Parallel Computing, 2002.

[12] Allcock, Bester, Bresnahan, Foster, Gawor, Insley, Link, Papka, *GridMapper: A Tool for Visualizing the Behavior of Large-Scale Distributed Systems*, Proceedings of High Performance Distributed Computing **11**, Edinburgh, Scotland, 2002.

[13] T. Fredian and J. Stillerman, *MDSplus Remote Collaboration Support – Internet and the World Wide Web*, Fusion Engineering and Design **43**, (1999).

[14] HDF5 User Manual, http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.intro.html.

[15] Ma, Winslett, Lee, Yu. Improving MPI-IO Output Performance with Active Buffering Plus Threads. International Parallel and Distributed Processing Symposium (IPDPS'03), April 22 - 26, 2003, Nice, France.

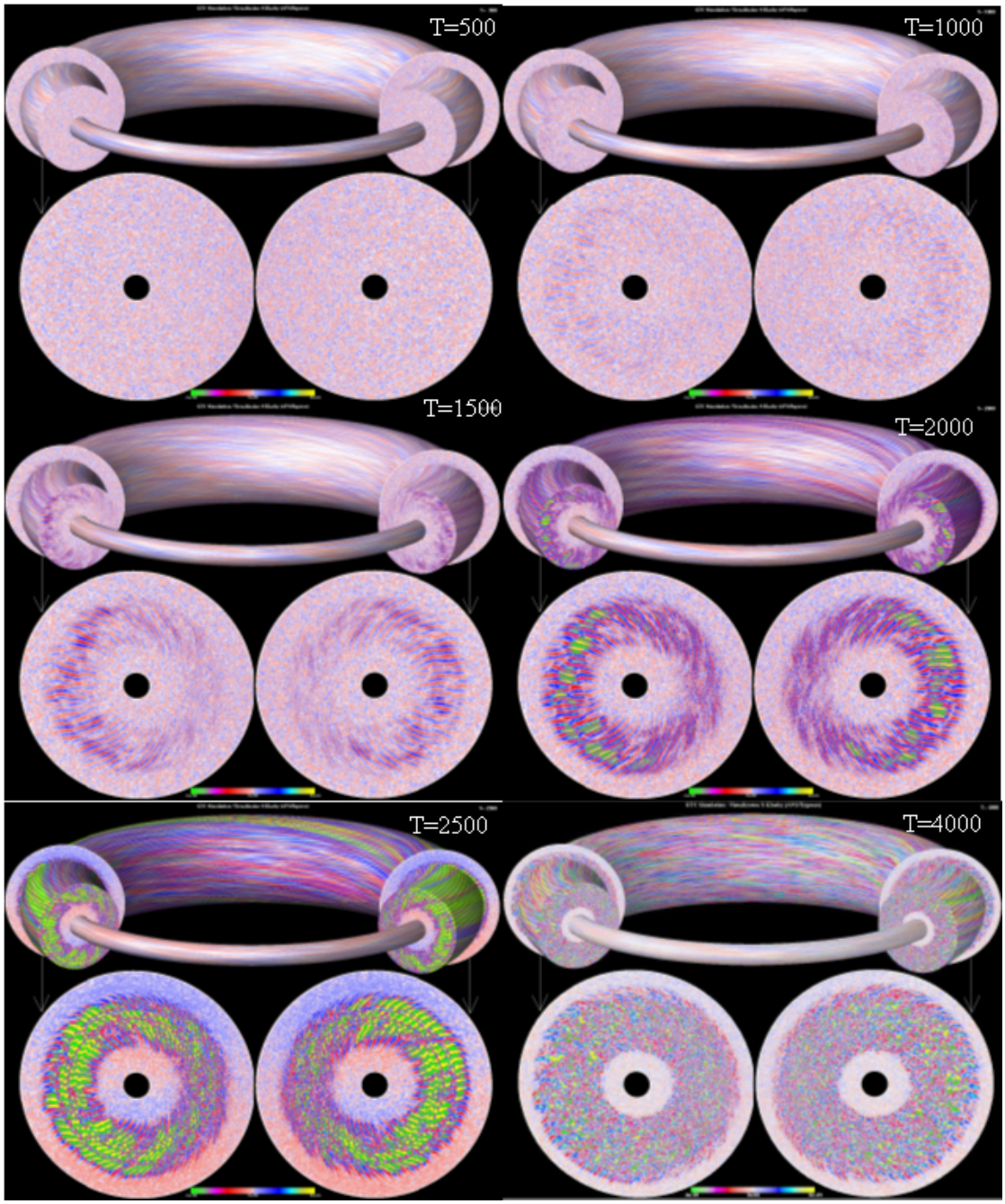

Figure 10 . A "serial" visualization of the GTC code from a streaming data experiment. This run was produced 700GB of data during a 72 hour run. The electrostatic potential is shown at different times during the simulation. At T=500, the potential still has a uniform distribution of turbulence. Later in time, we see coherent structures form in the potential. These finger-like structures act as energy dissipating channels for the energy of the system. As a second instability develops the flow becomes stochastic again.