Speeding up NGB with Distributed File Streaming Framework

Bingchen Li[1], Kang Chen[1], Zhiteng Huang[1], Hrabri L. Rajic[2], Robert H. Kuhn[2]

[1]Intel China Research Center Ltd.,
Haidian District, Beijing 100080 China
{bingchen.li, kang.chen, zhiteng.huang}@intel.com

[2]KSL, Software Products Division, Intel,
Champaign, IL 61820 USA
{hrabri.rajic, bob.kuhn}@intel.com

Abstract

Grid computing provides a very rich environment for scientific calculations. In addition to the challenges it provides, it also offers new opportunities for optimization. In this paper we have utilized DFS (Distributed File Streaming) framework to speed up NAS Grid Benchmark workflows. By studying I/O patterns of NGB codes we have identified program locations where it is possible to overlap computation and data workflow phases. By integrating DFS into NGB, we demonstrate a useful method of improving overall workflow efficiency by streaming the output of the current process to make an input of the following stage, reducing a workflow to a series of distributed producer consumer stages. DFS framework eliminates file transfers and in the process makes process scheduling more efficient, leading to overall performance improvements in the turnaround time for HC (Helical Chain) data flow graph under Globus grid environment with the embedded DFS over the original version of the benchmark.

1. Introduction

As a data flow graph encapsulation of NPB [1] modules, NGB (NAS Grid Benchmark) [2] provides a paper-and-pencil specification of a benchmark suite for computational grids. The benchmark purpose is to rate performance of target Grid system via a set of data flow graphs of CFD (Computational Fluid Dynamics) problems. Through investigation of NGB I/O with Intel® Trace Analyzer and Collector (former Vampir tools) [3], we have identified substantial output and input file I/O activity to motivate data optimization code modification. Scaling the benchmarks improves the scheduling overheads, but increases the data size necessitating proportionally more time spent on file I/O and its movement making the savings very attractive. Our analysis results corroborate previous findings [4] that NGB suite spends some unnecessary time on scripts and task scheduling. The smaller the problem size is, the greater the scheduling overhead results from job management [5]. To obtain the overall speedups the total turnaround time is used as a benchmark metrics in our experiments.

DFS (Distributed File Streaming) [6] is an I/O optimization framework, that streams file I/O output of a data producer program to a data consumer program. The faster data transfer over the interconnect replaces the file I/O write and read stages and a file transfer. In a typical data dependent workflow, such as the HC workflow of NGB [2], computing tasks first generate one or many output files on the local storage and then these files are transferred by FTP-like utilities to the remote nodes. Only after all the output files have been transferred, the next stage computing tasks on the remote node begin to commence. DFS framework has the capability to deliver multiple files to multiple nodes concurrently in an asynchronous manner. It uses parallel TCP streams like GridFTP [7] to maximize the network bandwidth utilization. It reuses TCP channels in a non-blocking mode to lessen the system load and reduce data transfer overhead. More about DFS would be available soon [6].

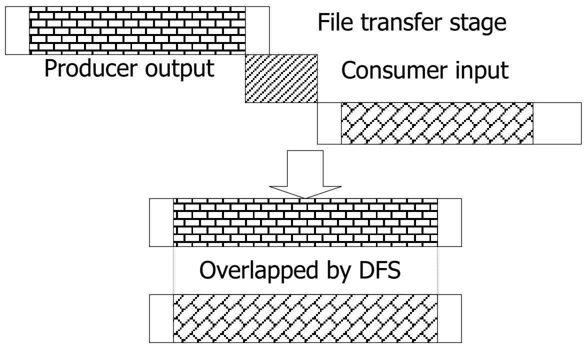


Figure 1. Distributed I/O overlapping by DFS

In Section 2 of this paper, we describe the overall structure and the important components of DFS. We focus on the inter-process data exchange (shared-memory) and the file transfer mechanism to depict how DFS can optimize distributed application I/O.

In Section 3, in addition to doing benchmarking of our hardware and software environment, we give comprehensive characterizations of the NGB I/O patterns.. We also introduce DFS I/O speed-up model in this Section.

We give the details of the optimizing and tuning work done in Section 4, where we describe our Grid environment. The DFS integration steps into NGB and the experiment results are presented in Section 5.

In the Section 6 we summarize the work explaining the impact DFS has on the distributed computing, especially on data-intensive applications.

2. Distributed file streaming

DFS is an application level framework providing data streaming capability to embed into distributed computational tasks. It provides an optimized transport layer to connect distributed nodes, a POSIX compliant I/O API, flexible data caching and buffering, and other mechanisms that guard against network congestion and network unavailability. The framework is split into producer and consumer parts. It includes a daemon program, file I/O API library, and auxiliary components on each side, Fig. 2.

2.1. DFS daemons

Depending on the source and destination of a data transfer process, DFS producer and consumer side daemons have different functionalities for data handling, including buffering and caching. For every file I/O that is redirected via DFS a new connection is established.

When the data is transferred from one machine to another via DFS, the data sender program needs to establish control communication with the local DFS daemon first and then start delivering data through application process space DFS API. DFS daemons establish shared memory buffers that are used to exchange data with the user application. If the data size exceeds shared memory buffers capacity, the local disk space will be used as a cache to store overflow data. That could impact the DFS I/O performance reducing it to the disk bandwidth rates. That would also mean that the consumer could not keep up with the producer I/O rate, so not much would be lost. For large problems virtual memory paging could evict pages from the shared memory buffers, producing impact on the DFS performance.

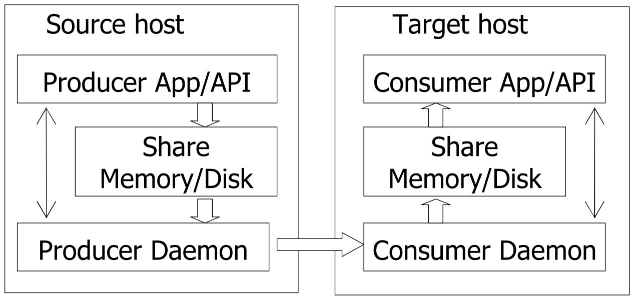


Figure 2. DFS Daemons and data exchange channel

The producer daemon manages shared memory buffers and controls the data flow from user applications. It establishes a new TCP connection with the remote consumer daemon whenever there is a new file redirected data added into the data task queue. In the same way one producer daemon may connect to multiple consumers if data from a single file is delivered to multiple remote destinations.

A similar, almost identical scenario happens on the consumer side except for the fact that streaming data flows in the opposite direction. Data is buffered in the shared memory. It can optionally be saved into the local file system if a permanent backup is required.

DFS daemons maintain two kinds of communication control links, one between remote daemons, and the other between a local daemon and the application. Parallel TCP streams, reusable TCP channels, and non-blocking socket communication are used to maximize network bandwidth utilization and to minimize system overhead associated with the network communication layers.

2.2. DFS file APIs

Considering that the vast majority of scientific computing applications and legacy applications use standard POSIX file manipulation semantics, we have decided that DFS APIs be compliant with the POSIX file I/O semantics and the calling sequence. The set of APIs on both consumer and producer are summarized in Table 1.

Table 1. Comparison of DFS APIs with POSIX's

POSIX	DFS Producer	DFS Consumer
open/create	shm_pdh_open	shm_cdh_open
read		shm_cdh_read
write	shm_pdh_write	
close	shm_pdh_close	shm_cdh_close

Shm stands for shared memory, pdh for the producer data handler, while cdh for the consumer data handler.

To make the interface usable for codes written in FORTRAN language, DFS API has a wrapper interface available according to the FORTRAN function call conventions.

2.3. DFS data streams

We have defined message packets and communication semantics between daemons to support both parallel and striped file transfers. To support the data streaming mode; we are using a ‘push’ mode, we have defined three size-variable message types: ‘MSG_OPEN’, ‘MSG_CREAT’, and ‘MSG_STORE’ for transfers from the producer side to the consumer side. Also in the set is the fixed-size message type ‘MSG_CLOSE’.

A typical data stream from producer daemon to consumer daemon has the message order as depicted in Figure 3.

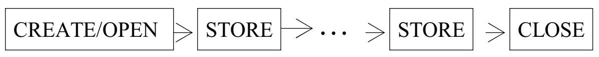


Figure 3. A typical data stream from producer daemon to consumer daemon

3. Experiment environment and NGB code Evaluation

We describe the performance of our computational environment in section 3.1 by presenting Iozone [8] and Iperf [9] benchmark run results. We also give GridFTP performance numbers in section 3.2. These numbers will be utilized in the following sections.

Before the integration of the DFS framework into the NGB suite, we did some pre-investigation work to determine the NGB codes I/O timelines to estimate possible impacts of DFS optimization on NGB execution times. These data are presented in section 3.3. The original and DFS enabled NGB I/O performance models are presented in Section 3.4.

3.1 Hardware benchmarking

In addition to the CPU performance, disk throughput and network bandwidth are the other two important factors that have direct effect on NGB running times. Running the Iozone suite has shown that our Ultra SCSI 320 disk can provide about 60MB/s read and write throughputs on EXT3 file system without data caching.

The full scale Iozone benchmark results are shown in Figure 4 and Figure 5 for file system write and read rates.

In the second set of experiments Iperf network bandwidth benchmark has obtained 119MB/s bandwidth rates on our 1 GbE network over the TCP protocol.

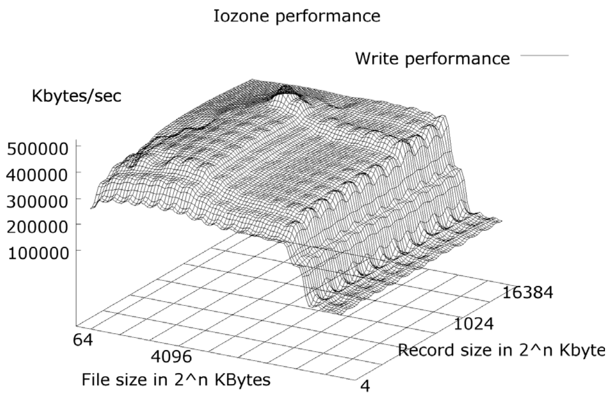


Figure 4. File system write performance

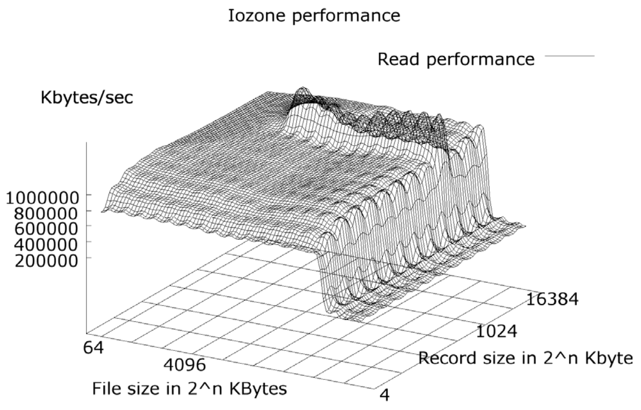


Figure 5. File system read performance

3.2 GridFTP performance

To evaluate GridFTP performance we used an internal test suite to measure the CPU loads and the network bandwidth utilization starting with a single TCP connection and all the way up to 10 TCP connections. File sizes ranged from 1MB to 1GB. Since we were not able to get good parallel data transfers in our environment, due to being bounded by non RAID disk I/O throughput, Figures 8 and 9 only show network bandwidth utilization of a single TCP connection.

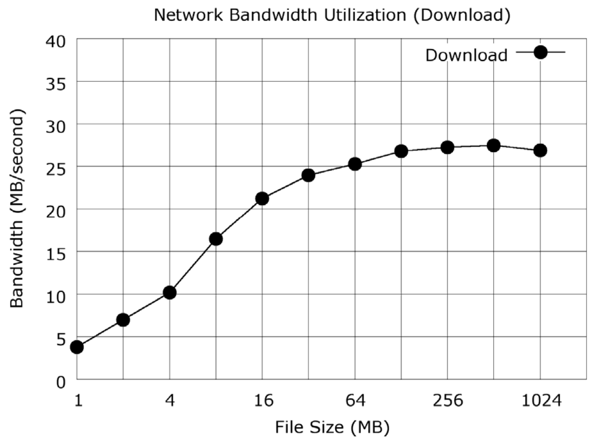


Figure 6. GridFTP bandwidth utilization (download)

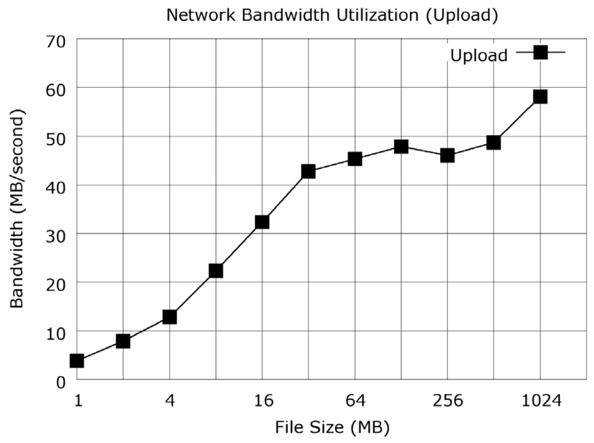


Figure 7. GridFTP bandwidth utilization (upload)

We also measured the average RTT (round trip time) by measuring the time by sending thousands of 0-byte files. As expected, the DFS protocol overhead is much smaller than the overhead of the general purpose HTTP, FTP and GridFTP utilities (See Table 2.) When large files are transferred, GridFTP overheads get dwarfed in comparison with the total running times, implying that the smaller size problems are not reliable indicators for performance improvement studies.

Table 2. Round trip time

Protocol	RTT (second)
HTTP	0.01263
FTP	0.02360
GridFTP	0.30452
DFS	0.002163

The call graph information from Vtune [10] has shown that GridFTP spends some time in the security related layers. We have measured the time distribution in each shared library when uploading 1KB size file by GridFTP.

Table 3. Top 5 time-consuming GridFTP libraries

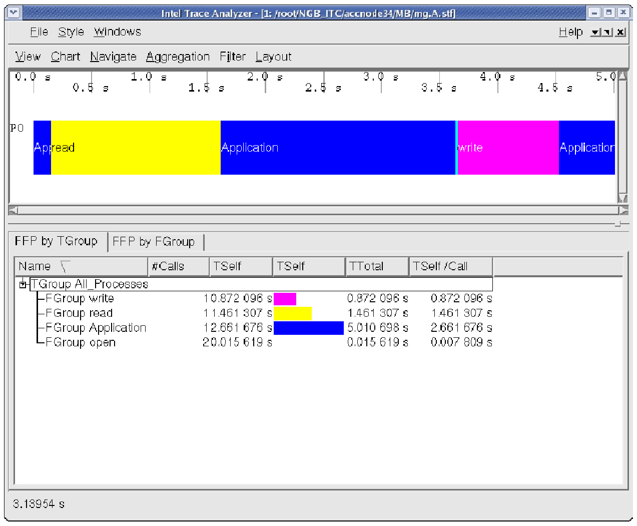
Time (μs)	Percentage	Package Name
231108	65.32%	libcrypto_gcc64.so.0
98297	27.78%	libc.so.6.1
7663	2.17%	libglobus_common_gcc64.so.0
3438	0.97%	Libssl_gcc64.so.0
3292	0.93%	libglobus_xio_gcc64.so.0

DFS overhead should be looked in the proper light. It is not something that we advocate to remedy in GridFTP, since this overhead becomes negligible for larger file size transfers.

3.3 I/O characterization of NGB benchmarks

To estimate the improvement that can be expected from using the DFS framework, we needed to have I/O profiles of the NGB codes. We have used Intel® Trace Collector and Analyzer instrumentation software to obtain such information. ITC API calls were inserted into NGB Fortran source code to generate time stamps of the I/O function calls during the execution. Figure 8 is a screenshot of an actual ITA run on our system. By using the ITC trace output, we were able to calculate the I/O intervals and measure the I/O calls position of the each NGB task accurately. The continuous time fragments in Figure 8 illustrate the time spent in the read/write/open calls and in the whole application.

After merging the ITC outputs from all of the compute nodes, we were able to reconstruct the benchmark timelines. From the results of I/O measurements we were able to calculate the I/O portion in each NGB data flow graph and how much improvement we can gain via using DFS framework. Within these 4 data flow graphs (ED, HC, VP and MB) of NGB, HC is the best suitable candidate for the DFS embedded optimization. ED benchmark is embarrassingly parallel, without any dependencies, VP has unequal data paths leading to stages 5 and 8, limiting performance gains, while MB benchmark has multiple dependencies, with limited I/O in the first parallel stage. Figure 9 illustrates the timeline of the HC (Helical Chain) modules and shows overlapped I/O regions.



Name	#Calls	TSelf	TSelf	TTotal	TSelf /Call
TGroup All_Processes					
FGroup write	1	0.872 096 s		0.872 096 s	0.872 096 s
FGroup read	1	1.461 307 s		1.461 307 s	1.461 307 s
FGroup Application	1	2.661 676 s		5.010 698 s	2.661 676 s
FGroup open	2	0.015 619 s		0.015 619 s	0.007 809 s

Figure 8. I/O duration measurement by using ITA

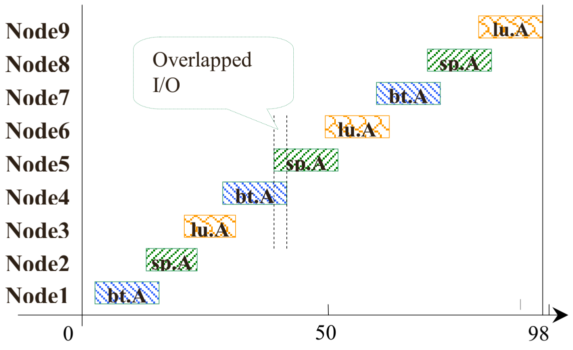


Figure 9. Time line of DFS enabled HC code

For the class A of the HC benchmark the I/O timelines show that 10% of the total time could be eliminated by data streaming.

3.4 DFS speed-up estimation of the HC workflow

In the scientific benchmarks, where the computation steps are followed with data intensive I/O sections it is possible to fully utilize the network or the file I/O system. In this case the I/O stage durations could be easily determined by the amount of data that needs to be handled.

Besides class W, that involves mesh filter due to different input and output matrix sizes, there are NGB class S, A, B, and C data sets that are of our interest. For the HC program, floating point and I/O data sizes are listed in Table 4.

Table 4. Problem size and data size of HC

Class	Tasks	Double Float	Data Size (bytes)
S	9	10,140 * 8	648,960
A	9	1,352,000 * 8	86,528,000
B	18	5,410,590 * 17	735,840,240
C	27	21,520,890 * 26	4,476,345,120

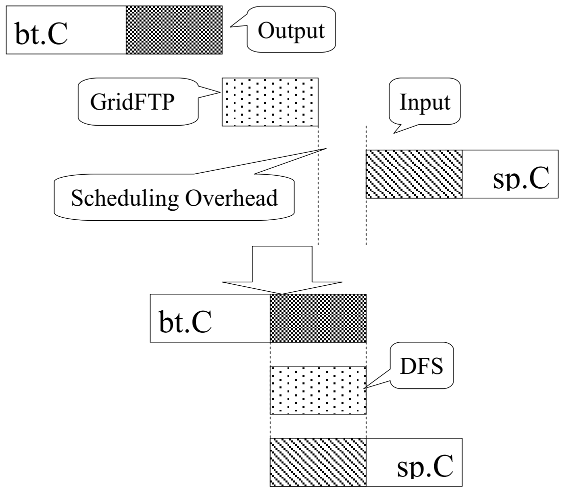


Figure 10. Theoretical DFS speed-up model for two NGB I/O coupled modules.

From Figure 10 we can see that the I/O activity between two modules includes file output, file input, GridFTP file transfer time, and the scheduling overhead. The scheduling overhead is the time between the end of file transfer and the beginning of the computation phase and involves waiting for the file to arrive:

```
While  [$NUM_INPUTS_RECEIVED  lt  $NUM_INPUTS]
do
Sleep 1
…….
```

From the above code section it could be seen that the maximum scheduling overhead between two tasks is less than one second.

Based on above measurements and data sizes, the I/O duration of the HC class C benchmark can be expressed as:

$$\frac{D}{S_w}+\frac{D}{S_r}+\frac{D}{S_g}+O_s,$$

Where D is data size, $S_{r,w,g}$ stands for read, write, and GridFTP data transfer speed. O_s is the scheduling overhead. Upon substitution the I/O duration comes to:

$$\frac{4.5GB}{60MB/s}+\frac{4.5GB}{60MB/s}+\frac{4.5GB}{45MB/s}+26s=276s$$

Actually, after considering I/O caching, the real I/O duration is shorter than 276 seconds.

After DFS integration the HC I/O duration can be calculated as:

$$\frac{D}{S_d} = \frac{4.5GB}{119MB/s} = 38s,$$

where S_d stands for data transfer speed of DFS. Based on above formulae and the calculations, DFS could theoretically save, at the best, 238 seconds of the total I/O time for the class C problem.

It should be noted that this is not a fully realistic situation, just a useful model. If the producer and consumer are pretty well matched in I/O write and read operations respectively or there is enough buffer space to offset that, as this is true in our case there is no need to cache data on the consumer disk which could degrade the bandwidth from 119 MB/s to 60MB/s. Also not investigated is an impact of virtual memory paging when evicting shared memory DFS buffers.

4. Computational Environment and DFS Optimizing and Tuning

We have done runs on an IA64 cluster. All machines were connected with 1 gigabit Ethernet. Every node was a 2-way SMP machine, equipped with two 1.5GHz Itanium2 processors and 2GB physical memory. Each node had one 36GB Ultra SCSI 320 hard disk which was used as a local storage.

One of the nodes was used as launcher while the other nodes were used as computing nodes. We installed Globus toolkit version 4.0 [11] as our Grid middleware. The installed OS was RedHat Enterprise Linux 4.0 (with 2.6 kernel). We have used updated Intel® C/Fortran compiler 9.0, with "-O3" optimization level to build NGB programs and the DFS framework [12]. Table 5 compares GNU gcc/g77; the original makefiles have specified "-O3" compilation optimization levels, and Intel® icc/ifort compiled HC runs.

Table 5. ICC and GCC compilation impact on HC turnaround times

Class	GCC 3.4.3	ICC 9.0
S	20	20
W	21	20
A	446	98
B	2008	475

It is expected that Intel compilers would have a better performance. For consistency, we prefer to have all of the stages of the workflow fully optimized. The side benefit was faster computations and a higher percentage of the I/O in the turnaround times. That produced higher DFS performance improvements, a fact that underscores the impact and importance of DFS framework on the workflow running times.

4.1 Optimization work on the Globus job manager

We have changed the polling frequency of Globus job manager from 10 seconds to 1 second in order to reduce the scheduling overhead, which was dominating the execution time for the small size benchmarks. The polling frequency was tested by submitting a command 'globus-job-run hostname /bin/sleep time'. Table 6 summarizes the impact of the changed polling interval.

Table 6. Response time for 10 and 1 second polling intervals

10 sec (before)		1 sec (after)	
Sleep Time	Real Time	Sleep Time	Real Time
0	0.800	0	0.725
1	10.895	1	1.774
2	10.870	2	2.888
5	10.915	5	6.216
11	21.053	11	11.754

4.2 DFS framework tuning work

To maximize the network bandwidth utilization, we have tuned several network parameters. Figure 11 illustrates the relationship of the internal buffer size and network bandwidth utilization for the DFS framework.

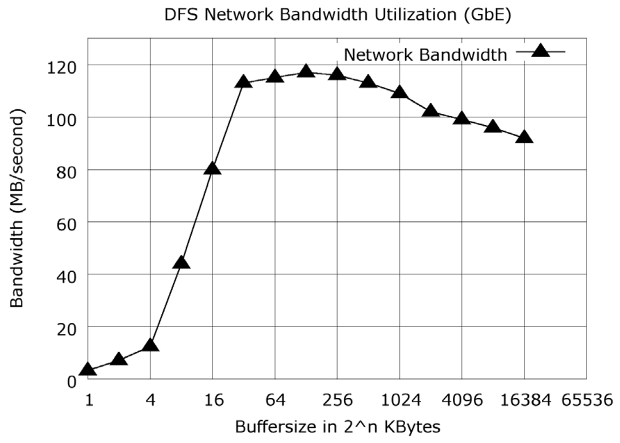


Figure 11. Network bandwidth vs. buffer size

The network bandwidth peak values are between 64KBs and 256KBs; small buffers do not use the TCP protocols efficiently, while large buffer sizes tend to increase DFS overheads for shared memory management and data traffic between the user applications and DFS daemons, so 128 KB buffer size works best.

5. Integrating DFS into NGB

The HC data graph flow has 3 computational modules: BT, SP and LU. We have located all the I/O API calls and have replaced them with the DFS API calls. The code section below is the data output part of the LU module after the DFS calls substitution.

```
   call dfs_phd_add_file_dest(outfile,
'OUTHOST1', retval)
   if ( retval .ne. 0 ) then
     print *, 'Failure: ……'
   else
     call dfs_pdh_open(dfs_handle,outfile,
retval)
     if ( retval .ne. 0 ) then
      print *, 'Failure: ……'
   else
      writelen = 5*(isiz1+1)*(isiz2+1)*isiz3
      call dfs_phd_write_double
(dfs_handle,u,writelen,retval)
   if ( retval .ne. writelen ) then
      print *, 'EOF reached'
   endif
   endif
   call dfs_pdh_close(dfs_handle)
   endif
```

We have also modified some points in ‘sparam.c’ file to make NGB runs support the largest data set, i.e. class C. ‘INHOST’ and ‘OUTHOST’ environment variables were added into ‘HC’ script to indicate file source and destination.

We have run the original and the modified HC code ten times to obtain reliable average turnaround times for each problem class. The results are presented in Table 7.

Table 7. Turnaround times for HC code

Class	Type	Time(seconds)	Improvement
S	GridFTP	20.1	21.39%
	DFS	15.8	
A	GridFTP	98	10.41%
	DFS	87.8	
B	GridFTP	474.6	9.84%
	DFS	427.9	
C	GridFTP	2053.8	9.73%
	DFS	1853.9	

The benchmark results match our previous performance improvement estimates. It is worth noting that a modified version of the benchmark with proportionally more I/O would lead to a better DFS speedup and that the 10% comes from inherent code I/O limits and not DFS limitations. Looking at Figure 1., it is obvious that the longer the I/O intervals; i.e. larger patterned areas, the more I/O overlap there is, resulting in more better producer and consumer overlap and better DFS speedups. 21% speed-up for the class S is not very reliable indicator of the DFS impact because of the small problem size and the proportionally large scheduling and other overheads; see Table 3 for the unmodified code run profile.

6. Conclusion

It was shown that DFS provides a convenient framework for optimizing distributed computing tasks with data dependencies between the computational stages. 10% speedups have been obtained on the DFS enabled NGB code Helical Chain on class A, B, and C problems. In addition to I/O overlapping, additional DFS framework advantage, albeit small was the elimination of the job scheduling overheads between the remote tasks. From the result of HC class S we could see that the scheduling cost became a significant factor because of a small problem size leading to somewhat inflated performance advantages. In the ideal situation, DFS can make a chain of tasks fully overlapped if their inputs and outputs are closely coupled as in Figure 1. The more and/or the larger overlap of the I/O sections between the producer and the consumer the more effective the DFS framework becomes.

Acknowledgements

The authors would like to thank Eric Huang and Wenguang Chen for their comments during the early stages of this study.

References

[1] D. H. Bailey, E. Barszez, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederiekson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks", Intl. Journal of Supercomputer Applications, v. 5, no. 3 (Fall 1991), pp. 63- 73.

[2] R. F. Van der Wijngaart and M. A. Frumkin. “NAS Grid Benchmarks Version 1.0”. Technical Report NAS-02-005, NASA Advanced Supercomputing (NAS), NASA Ames Research Center, Mo_ett Field, CA, 2002.

[3] Intel® Trace Analyzer & Collector http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/index.htm

[4] M.A. Frumkin, Rob F. Van der Wijngaart. “NAS Grid Benchmarks: A Tool for Grid Space Exploration.” Cluster Computing 5, pp. 247-255, 2002.

[5] Peng, L.; See, S.; Song, J.; Stoelwinder, A.; Neo, H.K. “Benchmark performance on cluster grid with NGB”; Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004. 26-30 April 2004, Page: 275a.

[6] K. Chen, Z. Huang, B. Li, E. Huang, H. L. Rajic, R. H. Kuhn, W. Chen "Distributed File Streamer: A Framework for Distributed Application Data Coupling", manuscript in preparation
[7] "GridFTP Universal Data Transfer for the Grid", White paper, http://www-fp.globus.org/datagrid/deliverables/C2WPdraft3.pdf
[8] Iozone file system benchmark, http://www.iozone.org
[9] Iperf, TCP/UDP Bandwidth Measurement Tool, http://dast.nlanr.net/Projects/Iperf
[10] Intel® VTune™ Performance Analyzer http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm
[11] Globus Toolkit 4.0 http://www.globus.org/toolkit/
[12] Optimizing Applications with the Intel ® C++ and Fortran Compilers for Windows* and Linux* Updated for Intel ® Compilers 9.0 WHITE PAPER July 19, 2005 http://cache-www.intel.com/cd/00/00/21/92/219281_compiler_optimization.pdf