A Java Implementation of MPI-I/O-Oriented Random Access File Class in AgentTeamwork Grid Computing Middleware

Joshua Phillips, Munehiro Fukuda†, and Jumpei Miyauchi
Computing and Software Systems, University of Washington, Bothell
18115 Campus Way NE, Bothell, WA 98033
†Primary Contact: mfukuda@u.washington.edu

***Abstract*— MPI-I/O defines a high-level file interface that enables multiple ranks to share a random access file. It would be highly attractive to grid-computing users that files are automatically partitioned and transfered to remote sites where their jobs can access the files through MPI-I/O. We are currently implementing in the AgentTeamwork grid-computing middleware system a series of these file-handling features including: file partitioning into strides, stride distribution to multiple processes, stride access through our MPI-I/O-oriented random access file class, stride exchange among processes, and barrier synchronization support. Particularly focusing on a design of our random access file class, this paper presents an implementation and performance results of AgentTeamwork's file-partitioning and stride-maintenance schemes.**

I. Introduction

Needless to say, efficient file transfer to and handling at remote sites is one of the key features to be facilitated in grid computing when we particularly focus on parallel job execution at remote computing nodes. As part of the MPI-2 specification, MPI-I/O defines a high-level file interface not only to support file partitions and access patterns adapted to parallel computing but also used to interact with grid I/O systems [1], [2]. By allowing multiple processes to share a file pointer, MPI-I/O can ease file-based inter-process communication that actually provides us with a programming model different from conventional message passing but rather resembling to distributed shared memory or objects. Therefore, it is highly attractive to grid-computing users that all remote file operations can be achieved through MPI-I/O.

Such remote file operations can be efficiently carried out through a series of following four sub tasks: (1) file partitioning, (2) file transfer, (3) file consistency maintenance, and (4) file collection. In other words, instead of duplicating and delivering an entire file to multiple processes, middleware systems should be able to partition a file into strides, each then automatically transfered to the corresponding process, exchanged with the other processes in a consistent order, and collected back to a user.

We are currently implementing these four file-handing features in the AgentTeamwork grid-computing middleware system [3]. Our GUI allows a user to instruct his/her file partitioning scheme to the system that then partitions a given file into strides, aggregates them based on the same destination, uses mobile agents to distribute the aggregated strides to remote sites, permits a remote process to access strides through our MPI-I/O-oriented random access file class (named *RandomAccessFile*), exchanges strides among different processes in support with barrier synchronization, and charges mobile agents with returning file outputs back to the user.

Among all these features, this paper brings its focus on file partitioning and stride maintenance through our Java implementation of *RandomAccessFile*. The rest of the paper is organized as follows: section 2 gives a system overview of AgentTeamwork; section 3 focuses on file partitioning; section 4 explains our file-stride maintenance; section 5 shows *RandomAccessFile*'s preliminary performance; and section 6 presents our conclusion.

II. AgentTeamwork

A. System Overview

AgentTeamwork is a grid-computing middleware system that coordinates parallel and fault-tolerant job execution with mobile agents [3]. A new computing node can join the system by running a UWAgents mobile-agent execution platform to exchange agents with others [4]. The system distinguishes several types of agents such as commander, resource, sentinel, and bookkeeper agents, each specialized in job submission, resource selection, job deployment and monitoring, and job-execution bookkeeping respectively.

A user submits a new job with a commander agent that receives from a resource agent a collection of remote machines fitted to the job execution. The commander agent thereafter spawns a pair of sentinel and bookkeeper agents, each hierarchically deploying as many children as the number of the remote machines. Each sentinel launches a user process at a different machine with a unique MPI rank, takes a new execution snapshot periodically, sends it to the corresponding bookkeeper, monitors its parent and child agents, and resumes them upon a crash. A bookkeeper maintains and retrieves the corresponding sentinel's snapshot upon a request. Input files and the standard input are packetized in inter-agent messages and delivered from the commander agent to each user process through a hierarchy of sentinel agents, whereas output files and the standard output are directly returned from each sentinel to the commander agent.

B. Programming Model

A user program is wrapped with and check-pointed by a user program wrapper, one of the threads running within a sentinel agent. The wrapper facilitates three libraries such as *GridTcp*, *mpiJava* [5], and *GridFile*. GridTcp implements error-recoverable TCP communication over multiple clusters. All *mpiJava* functions have been re-implemented with GridTcp to realize multi-cluster communication as in MPICH-G2 and to even tolerate cluster crashes [6]. GridFile provides the same interface as Java files including RandomAccessFile and buffers file contents as serializable data in the wrapper. A user program can take advantage of these fault-tolerant features by inheriting the *AteamProg* class.

Figure 1 shows a Java application executed on and check-pointed by AgentTeamwork. Besides all its serializable data members (lines 3-4), the application can register local variables to save in execution snapshots (lines 33-34) as well as retrieve their contents from the latest snapshot (lines 28-29). At any point of time in its computation (lines 12-24), the application can take an on-going execution snapshot that is serialized and sent to a bookkeeper agent automatically (line 22). As mentioned above, it can also use Java-supported files and mpiJava classes whose objects are captured in snapshots as well (lines 13 and 16).

Focusing on *RandomAccessFile*, a file can be shared among all processes, while each stride is actually allocated to and thus owned by a different process. Figure 1 assumes that each process owns and thus writes its $rank$ to a one-byte stride (lines 17-18) that is read by another process with $rank - 1$ upon a barrier synchronization (lines 19-20).

```
1  import AgentTeamwork.Ateam.*;
2  public class MyApplication extends AteamProg {
3    private int phase;
4    private RandomAccessFile raf;     // RandomAccessFile
5    public MyApplication(Object o){} // system reserved
6    public MyApplication( ) {         // user-own constructor
7      phase = 0;
8    }
9    private boolean userRecovery( ) {
10     phase = ateam.getSnapshotId( );// version check
11   }
12   private void compute( ) {         // user computation
13     raf = new RandomAccessFile(     // create a file
14               new File(''infile''),// input file
15               ''rw'' );             // mode
16     int rank = MPI.COMM_WORLD.Rank( );
17     raf.seek( rank );               // go to my stride
18     raf.write( rank );              // write my rank
19     raf.barrier( );                 // sync with others
20     int data = raf.read( );         // read my rank + 1
21     raf.close( );                   // close
22     ateam.takeSnapshot(phase);      // check-pointing
23     ...;
24   }
25   public static void main( String[] args ) {
26     MyApplication program = null;
27     if ( ateam.isResumed( ) ) {     // program resumption
28       program = (MyApplication)
29          ateam.retrieveLocalVar( ''program'' );
30       program.userRecovery( );
31     } else {                        // program initialization
32       MPI.Init( args );             // javaMPI invoked
33       program = new MyApplication( );
34       ateam.registerLocalVar( ''program'', program );
35     }
36     program.compute( );             //now go to computation
37     MPI.Finalize( args );
38 } }
```

Fig. 1. File operations in AgentTeamwork's application

III. File Partition

MPI-I/O defines a set of file contents accessible to each rank as a file *view* that is tiled with a repetitive sequence of identical *filetype*s from an absolute byte *displacement* relative to the beginning of a given file. Each *filetype* consists of *etype*s and *hole*s, the former specifying data types accessible to the rank, whereas the latter presenting inaccessible locations. Figure 2 gives an example of four different views, each defined for ranks 0 to 3. For instance, ranks 0 and 1 start two-byte *filetype*s with displacement 0, each consisting of a one-byte *etype* and hole whose order is opposite to each other. Similarly, ranks 2 and 3 tile their file with four-byte *filetype*s, each consisting of a one-character/one-short etype[1] and a two-byte hole that appear as complementing each other's hole.

AgentTeamwork maintains each rank's file view using a data structure named *RAFPartitionInfo*. Figure 3 illustrates four *RAFPartitionInfo* structures, each respectively corresponding to ranks 0 through to 3's file view given in Figure 2. Assuming that a file view can include multiple *etype*s, *RAFPartitionInfo* includes a table of *etype* entries. Pointed to by a 0-based index, each entry describes a different *etype* that starts from a given *displacement* within its *filetype*, consists of a specific primary data *type*, and occupies a given *size* of bytes. In addition to an *etype* table, *RAFPartitionInfo* also maintains its corresponding *rank*, its *filetype*'s initial *displacement* from the beginning of the file, its *filetype* size as ($upperbound - lowerbound$), the *total bytes* of all its *etype*s as well as additional synchronization variables such as *barrier_status* and *collective_action*. (Note that the last two variables are detailed below in Section IV.)

When AgentTeamwork's GUI (named *SubmitGUI*) accepts a user's inputs regarding *filetype*s for all ranks, it automatically generates all per-rank *RAFPartitionInfo* instances and includes them in each rank's very first file stride as a header. Thereafter, *SubmitGUI* starts its file-partitioning work. A naive partitioning algorithm is to iterate a file scan for each rank where *SubmitGUI* extracts necessary *etype*s from the file upon every occurrence of the *filetype* that has been described in this rank's *RAFPartitionInfo*. Its obvious drawback is the more ranks the more repetitive scans of the same file.

To limit this scanning work to only once for each file, *SubmitGUI* derives a collection of *FiletypeToRanks* structures from *RAFPartitionInfo* instances. Each *FiletypeToRanks* entry maintains a different *filetype*, its initial *displacement* from the top of a file, and an array of elements, each presenting a distinct byte position in this *filetype* and recording all ranks that access this byte. Figure 4 gives an example to be generated from the four *RAFPartitionInfo*s defined in Figure 3. For instance, the first *FiletypeToRanks* entry maintains the *filetype* that starts its repetitive occurrence from the 10th byte position

[1]Our implementation is based on Java that allocates two bytes to a character data type.

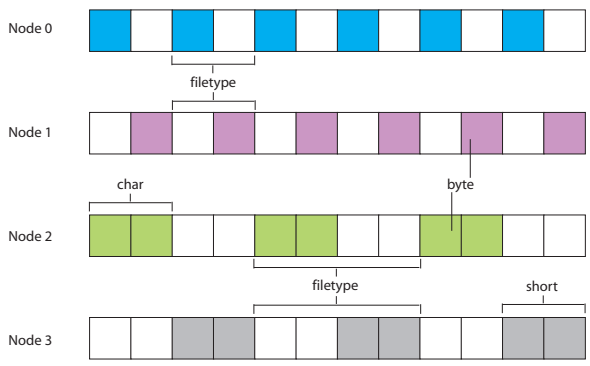

Fig. 2. An example of RandomAccessFile's partitioning scheme

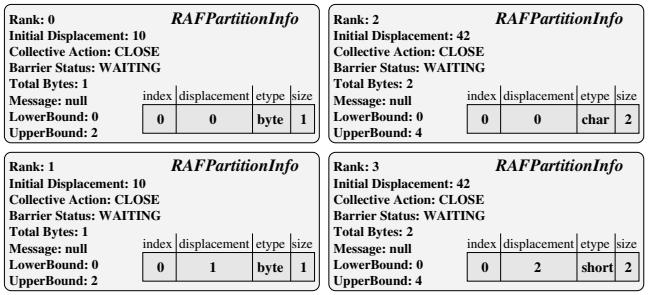

Fig. 3. RAFPartitionInfo structure entries

in a file and allocates two bytes, the first accessible to rank 0 and the second accessible to rank 1. The second entry starts from the 42nd byte position and occupies four bytes, the first two bytes accessed by rank 2 and the last two accessed by rank 3. The actual file scan is performed byte by byte as examining all *FiletypeToRanks* entries and their array of ranks, through which each byte is copied to appropriate rank-based file strides. Although this scanning complexity is yet proportional to the number of *FiletypeToRanks*, it is highly expected that a user program would define at most only a few different *filetype*s, (i.e., a few *FiletypeToRanks*, which thus restricts the overhead growth).

Upon generating file strides, *SubmitGUI* launches a commander agent that reads those strides, deploys a user job to remote sites through a hierarchy of sentinel agents, and thereafter keeps delivering each file stride through this hierarchy. Figure 5 illustrates this stride distribution. Given a

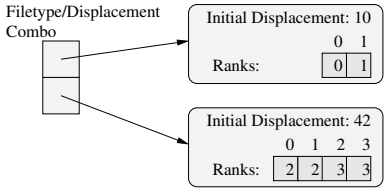

Fig. 4. FiletypeToRanks structure entries

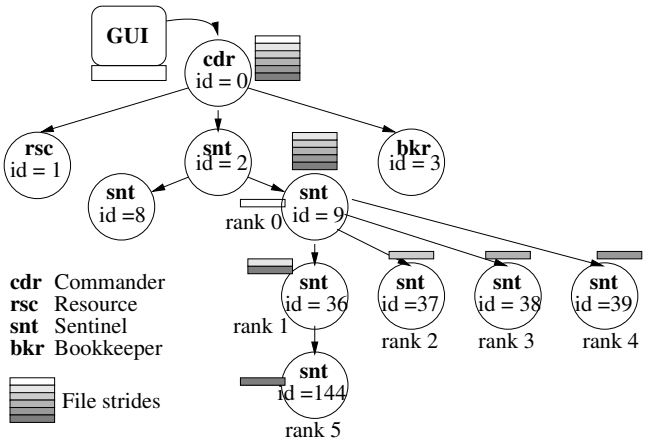

Fig. 5. Stride transfer through an agent hierarchy

hierarchy-unique identifier (abbreviated as an *id*), each sentinel determines an MPI rank for its user process, discovers all *id*s of its descendant agents, and similarly calculates their MPI ranks. With this knowledge, a sentinel agent can pass a file stride from its parent to the child agent whose descendant subtree includes this stride's final destination. To reduce inter-agent communication overhead, a sentinel aggregates and passes multiple strides at once to the same child agent.

IV. File Consistency

Using GridTcp and GridFile, we implemented AgentTeamwork's MPI-I/O-oriented random access file class. Our implementation was based on the following four strategies:

1) **Class interface**: We allow a user to handle AgentTeamwork's random access file as if s/he instantiated and used Java's *RandomAccessFile* class. We distinguish it from Java's as *RandomAccessFile-A*.
2) **Data allocation**: For each user process, the corresponding sentinel reconstructs a random access file locally by tiling the actual strides received from the commander and zero-initialized those owned by the other processes.
3) **Non-caching and write-through remote accesses**: Every read from a non-owned (and thus remote) stride uses a peer-to-peer GridTcp socket to transfer the stride directly to the requesting process. Similarly every write to a remote stride is immediately forwarded to the owner process. Needless to say, each access to the same stride is carried out exclusively.
4) **Barrier synchronization**: *RandomAccessFile-A* automatically makes all user processes synchronized together so as to make a consensus on resizing or closing a shared file. It also facilitates this barrier synchronization at a user level as *RandomAccessFile.barrier( )*.

RandomAccessFile.read( ) and *write( )* are implemented in a consistent manner as follows. The *read( )* function starts with scanning a collection of *RAFPartitionInfo*s to identify all ranks that share ownership in the range to be read (task 1). Thereafter, for *RAFPartitionInfo* of each rank identified, *read( )* must retrieve all *etype*s that cover the range to be

read (task 2), and read the file range overlapped with each *etype* retrieved (task 3). If an identified rank is local to the *read( )* call, tasks 2-3 are completed instantly. Otherwise, a read request is sent to a remote rank through a GridTcp socket. Each rank spawns a *RandomAccessFile* communication thread that exchanges a request and a response. Although a pair of threads perform tasks 2-3 respectively at a local and a remote side, a remote thread reads the data from its partition into a socket output stream while the local thread reads the data from a socket input stream into the user buffer. The *write( )* function works in the exact same sequence of those tasks except that the data flow is reversed.

Despite non-caching and write-through stride accesses, it is still anticipated that two or more user processes can read different file contents from the same remote stride, depending on various conditions that may interleave their read requests with a write request. The purpose of *RandomAccessFile.barrier( )* is to finish all on-going stride accesses and to allow all user processes to obtain identical stride contents in the conventional weak consistency model. Its implementation is straightforward in that each rank broadcasts a synchronization message to and receives one from all the others upon encountering a barrier. In addition to user-driven barrier synchronization, *RandomAccessFile-A* distinguishes three barrier types such *close*, *set_length*, and *length_inc*, each used to reach a consensus among all ranks in terms of closing the current file, resizing the file length, and increasing the file size. *RAFPartitionInfo* maintains the current barrier type and its progress in its *collective_action* and *barrier_status* variables respectively.

V. Performance Verification

We are currently conducting functional and performance verification of *RandomAccessFile-A* using a Giga-Ethernet cluster of 24 Dell computing nodes, each with 3.2GHz Xeon, 512MB memory and 36GB SCSI hard disk. We have prepared two test programs named *heartbeat* and *master-workers*. N processes are given the same *filetype* that includes N *etype*s, each composed of four doubles and owned by a different rank. Heartbeat engages each rank in modifying its own *etype*s and thereafter reading its neighboring rank's *etype*s, whereas master-workers makes rank 0 collect all *etype*s that the other ranks have modified. These access patterns represent typical spatial simulations and bag-of-task applications.

Figure 6 shows time elapsed to run these test programs as increasing the file size to 10MB and the number of processors to 16. While the time elapsed was proportional to the file size in terms of a two-processor execution, its growth rate gradually slowed down as the number of processors gets increased. Obviously this has been resulted from processor parallelism. However, the speed-up was quite small. For a 10MB file, 16 processors completed heartbeat and master-slaves only 2.3 and 1.7 times faster than two processors did. Furthermore, the absolute performance was inevitably slow. It took more than 270 seconds to have 16 processors exchange 10MB file strides in both tests. There are three reasons: (1) GridTcp redundantly copies in-transit messages into its rollback queue for recovery

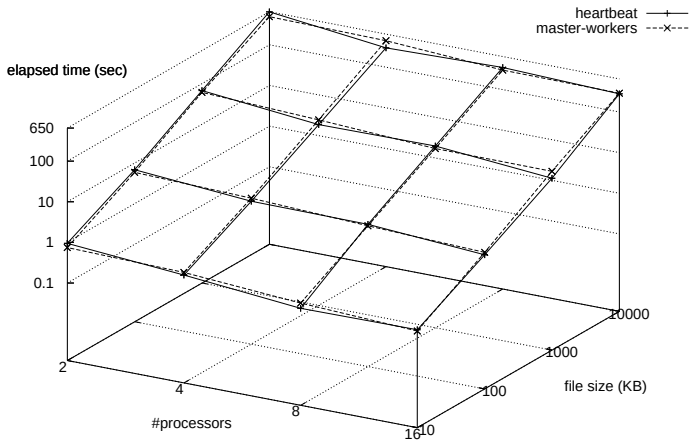

Fig. 6. Performance of remote stride accesses

purposes; (2) each *filetype* is transferred independently, which triggers thread context switches between RandomAccessFile and GridTcp; and (3) no remote stride is cached. Possible solutions include elimination of duplicated messages, stride transfer in a bulk, and stride caching.

VI. Conclusion

Based on the MPI-I/O concept, AgentTeamwork has enabled Java's *RandomAccessFile* to be partitioned into smaller strides, each delivered to the corresponding remote process and exchanged with the other processes in a consistent manner. Our next step is two-fold: (1) verifying job resumption accompanied with file strides at a different processor/cluster and (2) implementing fast bulk transfer and file stride caching.

Acknowledgment

This work is fully funded by National Science Foundation's Middleware Initiative (No.0438193).

References

[1] Message Passing Interface Forum, *MPI-2: Extention to the Message-Passing Interface*. University of Tenessee, 1997, ch. 9, I/O.

[2] A. Ching, K. Coloma, and A. Coudhary, *Challenges for Parallel I/O in GRID Computing*. Publisher's address: American Scientific Publisher, 2006, ch. 6, Grid I/O.

[3] M. Fukuda, K. Kashiwagi, and S. Kobayashi, "AgentTeamwork: Coordinating grid-computing jobs with mobile agents," *International Journal of Applied Intelligence*, vol. Vol.25, no. No.2, pp. 181–198, October 2006.

[4] M. Fukuda and D. Smith, "UWAgents: A mobile agent system optimized for grid computing," in *Proc. of the 2006 International Conference on Grid Computing and Applicaitons – CGA'06*. Las Vegas, NV: CSREA, June 2006, pp. 107–113.

[5] mpiJava Home Page, "http://www.hpjava.org/mpijava.html."

[6] M. Fukuda and Z. Huang, "The check-pointed and error-recoverable MPI Java library of AgentTeamwork gird computing middleware," in *Proc. IEEE Pacific Rim Conf. on Communications, Computers, and Signal Processing - PacRim'05*. Victoria, BC: IEEE, August 2005, pp. 259–262.