An Implementation of Parallel File Distribution in an Agent Hierarchy

Munehiro Fukuda [1] Jumpei Miyauchi [2]

[1] Computing & Software Systems, University of Washington, Bothell, *mfukuda@u.washington.edu*

[2] Computer Science, Ehime University, *miyauchi@koblab.cs.ehime-u.ac.jp*

Abstract

PC grid is a cost-effective grid-computing platform that attracts users by allocating to their massively parallel applications as many desktop computers as requested. However, a challenge is how to distribute necessary files to remote computing nodes that may be unconnected to the same network file system, equipped with insufficient disk space to keep entire files, and even powered off asynchronously.

Targeting PC grid, the AgentTeamwork grid-computing middleware deploys a hierarchy of mobile agents to remote desktops so as to launch, monitor, check-point, and resume a parallel and distributed computing job. To achieve high-speed file distribution, AgentTeamwork takes advantage of its agent hierarchy. The system partitions files into stripes at the tree root if they are random-access files, duplicates them at each tree level if they are shared among all remote nodes, fragments them into smaller messages if they are too large to relay to a lower tree level, aggregates such messages in a larger fragment if they are in transit to the same sub-tree, and returns output files to the user along multi-paths established within the tree. To achieve fault-tolerant file delivery, each agent periodically takes a snapshot of in-transit and on-memory file messages with its user job, and thus resumes them from the latest snapshot when they crash accidentally.

This paper presents an implementation and its competitive performance of AgentTeamwork's file-distribution algorithm including file partitioning, transfer, check-pointing, and consistency maintenance.

Keywords: parallel file distribution, fault tolerance, grid middleware, job deployment, mobile agents

1 Introduction

One of the main motivations for grid computing is to allocate to a massively parallel application as many computing nodes as requested. PC grid [1, 2] is such a cost-effective platform that attracts users by collecting clusters of research, instructional, and even office desktops. However, for data-intensive applications, PC grid needs to pay more attention to file availability than other platforms. This is because each computing node is necessarily not connected to the same network file system and in most cases is equipped with insufficient disk space to store entire files. Furthermore, each desktop computer may be powered on and off by its owner without any notification in advance, and thus it is not guaranteed that remote nodes stay available as long as a given job is running. Therefore, the key to PC grid implementation is to keep providing remote jobs with necessary file stripes in a timely fashion and to redirect file transfer to a new node where a crashed job has resumed its execution.

We have implemented a file-distribution algorithm to address these requirements in the AgentTeamwork system that deploys a hierarchy of mobile agents to remote sites in order to launch, monitor, check-point, and resume a parallel and distributed computing job [3]. Our algorithm focuses on two aspects of file access patterns: (1) all user processes may need the same file entirely and (2) they may access a different portion of the same file. To be fitted to the former aspect, our algorithm takes advantage of an agent hierarchy in AgentTeamwork by duplicating a file at each tree level. To be adjusted to the latter aspect, the algorithm partitions a file into stripes and delivers each one to a different process. (We define a file stripe as a collection of file blocks, each owned by the same process and appearing in a cyclic location of a given file.)

To distribute an entire file to each destination, we use fragmentation and pipelined transfer that allows a user process to advance its computation as far as only required file portions are made available. At the same time, file stripes do not have to be delivered independently. They can be aggregated in one message if they are in transit to the same sub-tree of descendant agents. Therefore, our file-distribution algorithm transfers a collection of user files through an agent hierarchy where each agent at the same tree level

divides files in smaller messages, aggregates them in a larger message headed to the same descendant tree, and duplicates them if necessary. Upon modification, files are returned back to the commander in smaller messages, traveling through the same hierarchy but actually relayed over every other tree layer to avoid traffic congestion toward the commander. (The following discussions use a file message as a transfer unit that has been made through file fragmentation and aggregation. We also distinguish partition and fragmentation as follows: the former divides a given file into stripes, whereas the latter divides a file, a stripe, or a file message into smaller file messages to transfer over network.)

To deliver each file stripe to a different process, our design is based on MPI-I/O that facilitates a high-level file interface to define file stripes and access patterns adapted to parallel-computing applications [4, 5]. Our GUI allows a user to instruct his/her file-partitioning scheme to the system that then partitions a given file into stripes, delivers them through an agent hierarchy to remote sites, permits a remote process to access a stripe through our MPI-I/O-oriented random-access file class (named *RandomAccessFile*), and exchanges stripes among different processes in support with barrier synchronization.

To achieve fault-tolerant file delivery, each agent maintains in-coming file messages in its local memory (or */tmp* disk), serializes them with an execution snapshot of its corresponding user process, and sends them to a different remote agent in their hierarchy, so that the user process can keep accessing the same files even upon a resumption from the very last snapshot. In other words, an agent keeps pumping file messages to the corresponding user process by resuming only lost messages (rather than replicates an entire file as performed in the conventional file replication.)

This paper presents an implementation and its convincing performance of file partitioning, stripe distribution, resumption, and consistency maintenance in AgentTeamwork. The rest of paper is organized as follows: Section 2 gives an overview of the AgentTeamwork system; Section 3 describes our file distribution, resumption, and maintenance strategies; Section 4 shows the performance of our implementation; Section 5 compares AgentTeamwork with related work; and Section 6 presents our conclusions as well as future plan.

2 AgentTeamwork

We start our discussions with an overview of the AgentTeamwork system, a basis of our parallel file distribution in an agent hierarchy.

2.1 System Overview

AgentTeamwork is a grid-computing middleware system that coordinates parallel and fault-tolerant job execution with mobile agents [3]. The system targets scientific-computing applications coded in Java because of two reasons: (1) Java is an architecture-independent interpretive language and thus its compiled programs are executable over heterogeneous grid-computing environments, and (2) Java's object serialization/de-serialization eases process check-pointing and error recovery, with which the system can keep resuming and running a user program at available (but different) computing nodes till its completion. Since many scientific applications tend to be written for parallel execution, AgentTeamwork assumes users who are familiar with MPI, MPI-I/O, or even sockets for inter-process communication and parallel file operations.

AgentTeamwork allows a new computing node to join the system by running a UWAgents mobile-agent execution daemon to exchange agents with others [6]. UWAgents maintains a parent-child relationship among mobile agents in a hierarchy that behaves similar to a process tree in operating systems but dynamically extends over network. Using this feature, AgentTeamwork deploys a user job to remote computing nodes with a several types of agents in a hierarchy. They are distinguished as commander, resource, sentinel, and bookkeeper agents, each specialized in job submission, resource selection, job deployment and monitoring, and job-execution bookkeeping respectively.

Figures 1 and 2 illustrate a job deployment in an agent hierarchy and its execution through agent interaction respectively.

First, a user submits a new job with a commander agent through AgentTeamwork's GUI (named *SubmitGUI*) or using some command line utilities. The commander agent migrates to a given XML resource database (named *XDBase*) and spawns a resource agent (*id 1* in Figure 2) that collects new

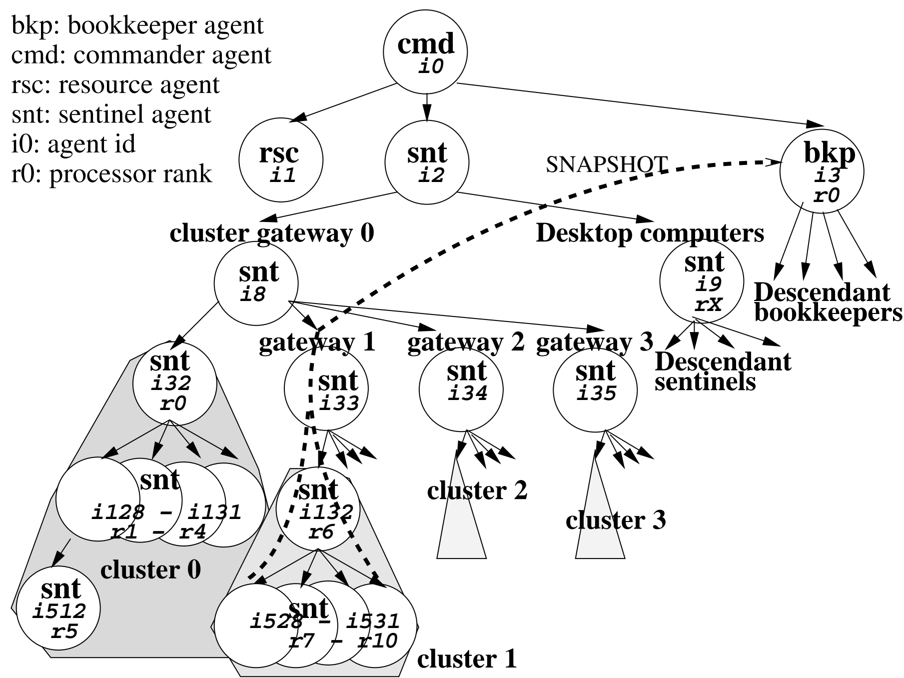

Figure 1. Job deployment by AgentTeamwork

computing-node information from a shared ftp server, registers it to the local XDBase, and retrieves a list of remote machines fitted to the job execution, (named a *resource itinerary*) [7].

The commander thereafter spawns a pair of agents, a sentinel with *id 2* and a bookkeeper with *id 3*, each hierarchically deploying as many children as requested in the resource itinerary. (Figure 2 shows only one child sentinel with *id 9* for simplicity.) If these computing nodes reside over multiple clusters, agents are deployed to a different cluster head or gateway where they further deploy a hierarchy of children to cluster-internal nodes [8]. Figure 1 shows an example where 11 child sentinels are deployed over clusters 0 and 1.

Before starting a user application, the commander, sentinels and bookkeepers exchange their IP address information through their hierarchy (with *talk(descendant_location)* and *talk(all_locations)* in Figure 2). Thereafter, each sentinel starts a user program wrapper with a unique MPI rank, (rank 0 through to 10 in the example) at a different machine. Launched from the wrapper, a user process periodically takes its computation snapshot and orders its local sentinel agent to send the snapshot to the corresponding bookkeeper (with *sendSnapshot( )* and *talk(save_snapshot)*). At every checkpoint all the user processes

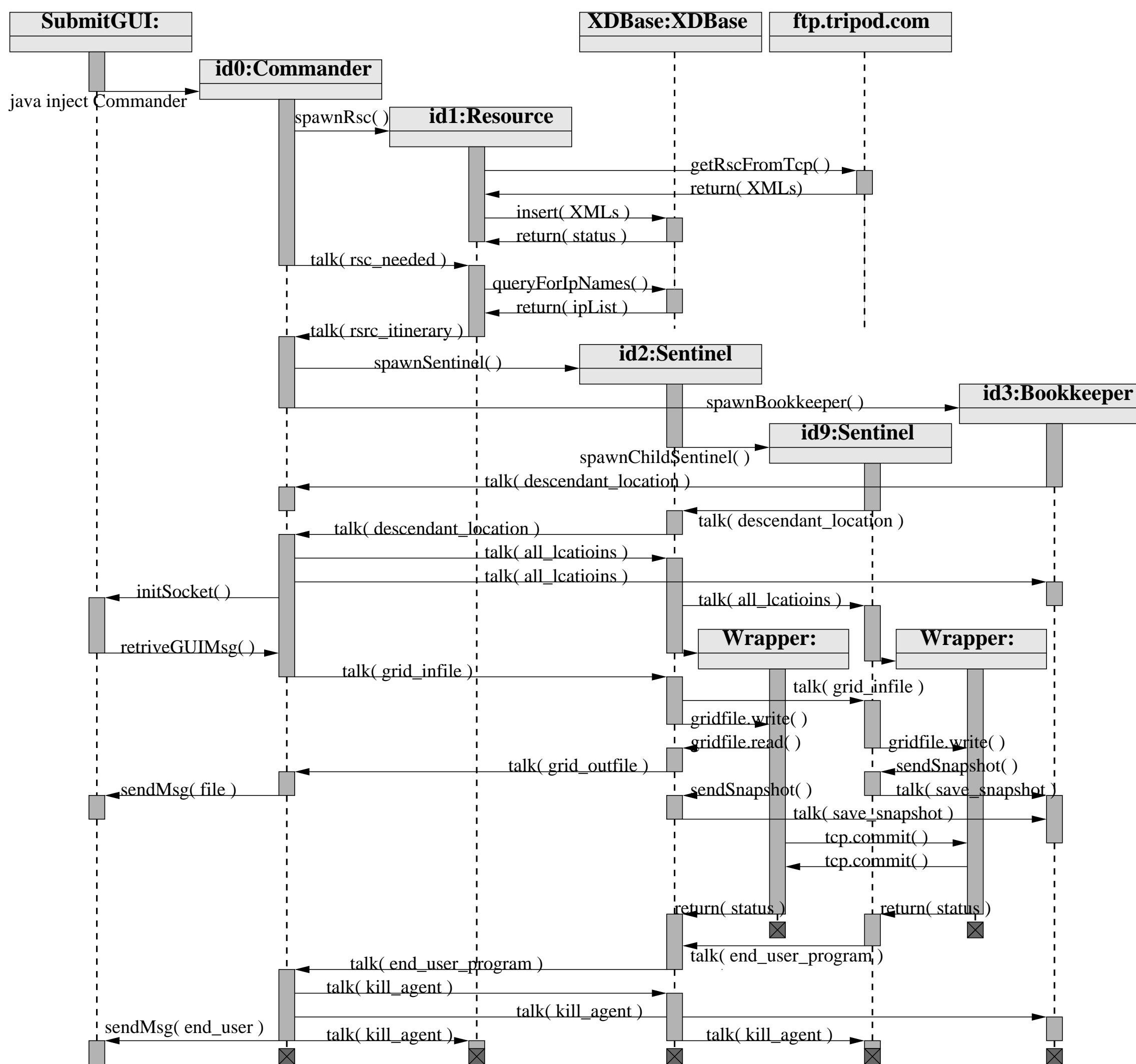

Figure 2. Job coordination through agent interaction

are automatically barrier-synchronized (with *tcp.commit( )*) so as to prevent any process from advancing too fast and thus from saving too many old messages in a snapshot. Each sentinel agent exchanges a ping and an acknowledgment with its children for error detection, and resumes them upon a crash. A bookkeeper maintains and retrieves the corresponding sentinel's snapshot on demand.

With *initSocket( )* and *receiveGUIMsg( )*, the commander establishes a socket to SubmitGUI and keeps receiving input files as well as the standard input in smaller packets. Repetitive calls of *talk(grid_infile)* relay these packets as inter-agent messages from the commander down to all user processes through their agent hierarchy. On the other hand, *talk(grid_outfile)* calls return output files and the standard output in

a reversed direction from each sentinel up to the commander agent.

Finally, a job termination is agreed among all agents with *talk(end_user_program)* and notified to SubmitGUI with *sendMsg(end_user)*, upon which they are terminated with *talk(kill_agent)*.

2.2 Programming Model

Figure 3 shows the AgentTeamwork execution layers from the top application level to the underlying operating systems. (Note that all components written in bold are AgentTeamwork's components and that all those in a shaded box are instantiated from a sentinel agent at each remote node.) The system facilitates inter-process communication with mpiJava [9] as well as Java Sockets for Java-based parallel applications. It also supports Java FileInputStream, FileOutputStream, and RandomAccessFile classes for remote file accesses. Since mpiJava has no MPI-I/O support, we have implemented the same concept in SubmitGUI and RandomAccessFile. The former assists a user in defining his/her file stripes with its GUI menus, whereas the latter allows an application program to access different file stripes with the *seek* method.

All these Java-based packages (including mpiJava) were re-implemented using *GridTcp* and *GridFile* in that we have developed fault-tolerant TCP and file manipulation. GridTcp monitors TCP links emanating from its local process, maintains the history of in-transit TCP messages, and restores all broken communication upon a process resumption. It also supports multi-cluster communication as in MPICH-G2 and tolerates cluster crashes [10]. GridFile implements an interface between a user program and AgentTeamwork by maintaining file contents in its error-recoverable queues.

Below these components is Ateam that provides a user application with check-pointing methods to explicitly take its computation snapshots through serialization of the application itself, GridTcp, and GridFile, all referenced to from the Ateam object. Ateam is then wrapped with a user program wrapper, one of the threads running within a sentinel agent. Recovered from a crash, the sentinel agent restarts its user program wrapper that resumes all the wrapped components from the latest snapshot.

The sentinel and the other AgentTeamwork's agents are executed on top of the UWAgents mobile agent execution platform which we have developed with Java as an infrastructure for agent-based grid

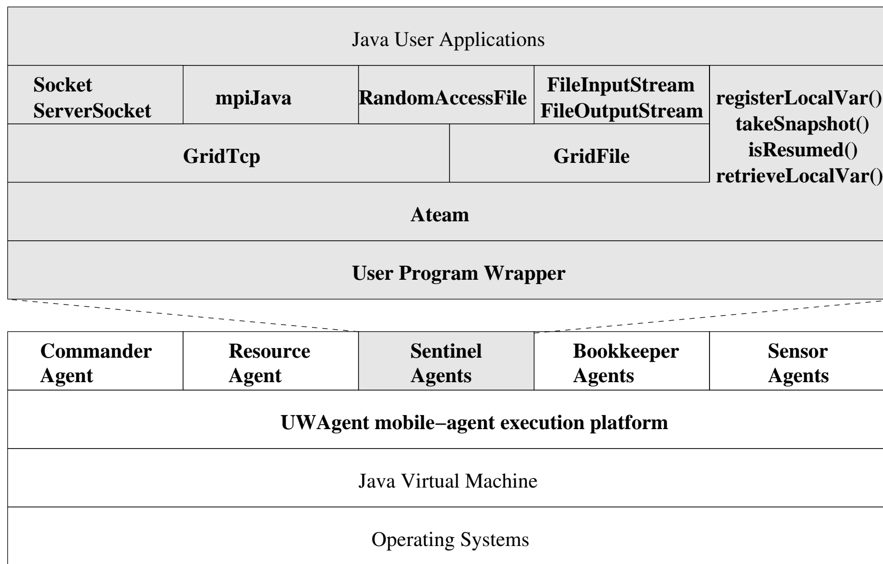

Figure 3. AgentTeamwork execution layer

computing.

Figure 4 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 38-39) as well as retrieve their contents from the latest snapshot (lines 32-33). At any point of time in its computation (lines 12-28), the application can take an on-going execution snapshot that is serialized and sent to a bookkeeper agent automatically (lines 22 and 26). As mentioned above, it can also use Java-supported files and mpiJava classes whose objects are captured in snapshots as well (lines 14, 16, and 24).

Focusing on RandomAccessFile, a file can be shared among all processes, while each stripe is actually allocated to and thus owned by a different process. Figure 4 assumes that each process owns and thus writes its $rank$ to a one-byte stripe (lines 17-18) that is read by another process with $rank - 1$ upon a barrier synchronization (lines 19-21).

3 File Distribution Strategies

The AgentTeamwork system delivers files to each remote process through four distinctive steps: (1) partitioning files into stripes, (2) transferring them through an agent hierarchy, (3) check-pointing them

```
 1  import AgentTeamwork.Ateam.*;
 2  public class MyApplication extends AteamProg {
 3    private int phase;                             // snapshot phase
 4    private RandomAccessFile raf;                  // RandomAccessFile
 5    public MyApplication(Object o){}               // the system-reserved constructor
 6    public MyApplication( ) {                      // a 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      ...;
14      raf = new RandomAccessFile(                  // create a RandomAccessFile object
15                new File("infile"), "rw" );
16      int rank = MPI.COMM_WORLD.Rank( );
17      raf.seek( rank );                            // go to my stripe
18      raf.write( rank );                           // write my rank
19      raf.barrier( );                              // synchronize with other ranks
20      int[] data = new int[1];                     // prepare a variable to store data
21      int data[0] = raf.read( );                   // read my rank + 1
22      ateam.takeSnapshot( phase++ );               // check-point intermediate computation
23      raf.close( );                                // close the RandomAccessFile object
24      MPI.COMM_WORLD.Reduce( data, 0,              // an MPI function
25              data, 0, 1, MPI.INT, MPI.Sum, 0 );// sum up what each rank has read
26      ateam.takeSnapshot( phase++ );               // check-point intermediate computation
27      ...;
28    }
29    public static void main( String[] args ) {    // start a user program
30      MyApplication program = null;
31      if ( ateam.isResumed( ) ) {                  // the program has resumed.
32        program = (MyApplication)                  // retrieve the latest snapshot
33           ateam.retrieveLocalVar( "program" );
34        program.userRecovery( );
35      } else {                                     // program has invoked from its beginning.
36        MPI.Init( args );                          // invoke mpiJava
37        program = new MyApplication( );            // create an application
38        ateam.registerLocalVar( "program",         // register my program
39                                program );
40      }
41      program.compute( );                          // now go to computation
42      MPI.Finalize( args );                        // end mpiJava
43  } }
```

Figure 4. File operations in AgentTeamwork's application

for recovery purposes, and (4) maintaining file-stripe consistency among different processes. The following subsections describe details of each implementation.

3.1 File Partitioning

File partitioning in AgentTeamwork is based on the MPI-I/O concept [4]. A user is supposed to instruct the system how to partition a given file into stripes and to allocate each to a different remote process by specifying the corresponding rank's *file view*, namely a repetition of an identical *filetype* that

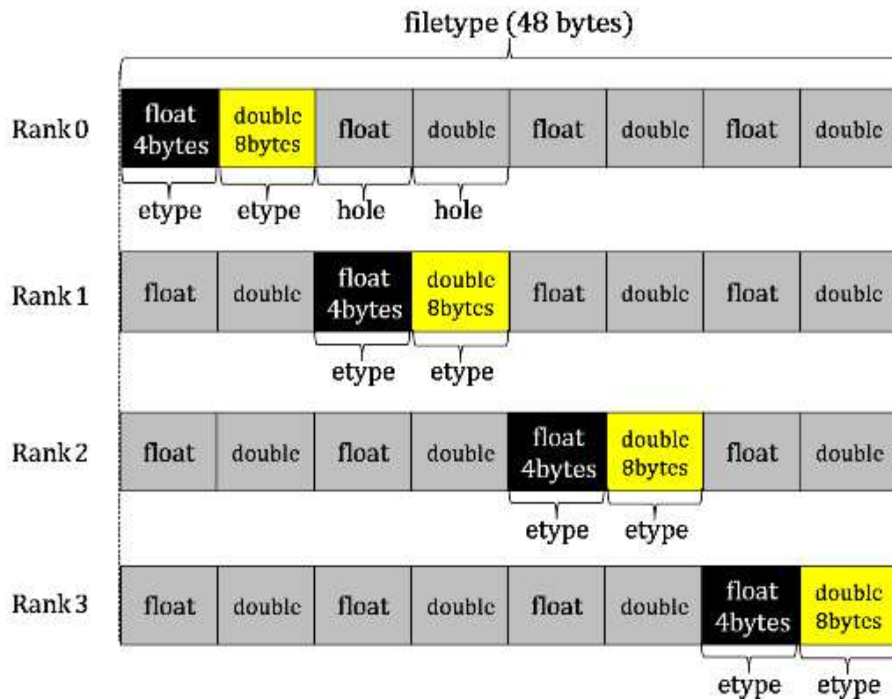

Figure 5. An example of MPI filetypes

is tiled with multiple *etype*s and *holes*. Figure 5 gives an example of four different *filetype*s, where each *filetype* allocates the $(i \times 2)th$ and $(i \times 2 + 1)th$ *etype*s (as a *float* and a *double* respectively) to rank i.

A user can define these *filetype*s through AgentTeamwork's SubmitGUI when submitting his/her job. Figure 6 captures a snapshot of SubmitGUI's job-submission window that makes the $0th$ and $1st$ *etype*s visible to rank 0. (Note that a user may skip this menu if s/he hopes to distribute a whole file to all ranks rather than partition a file in stripes, each delivered to a different rank.)

Upon receiving user inputs, SubmitGUI automatically generates a data structure named *RAFPartition-Info* for each rank. This data structure maintains a given rank's file view with the rank information, its *filetype*'s initial displacement from the beginning of the entire file, the *filetype* size in bytes, and a table of *etype*s that tile up this *filetype*. Each table entry is identified with its primary data type, the displacement from the top of the *filetype* it belongs to, the number of identical *etype*s tiled consecutively, and the *etype* size in bytes. Figure 7 illustrates four RAFPartitionInfo structures that respectively correspond to ranks 0 through to 3's file views given in Figure 5. For instance, rank 0's RAFPartitionInfo includes two table entries: the first defined as a *float*, starting from the $0th$ byte in the file view, tiled with only one *etype*

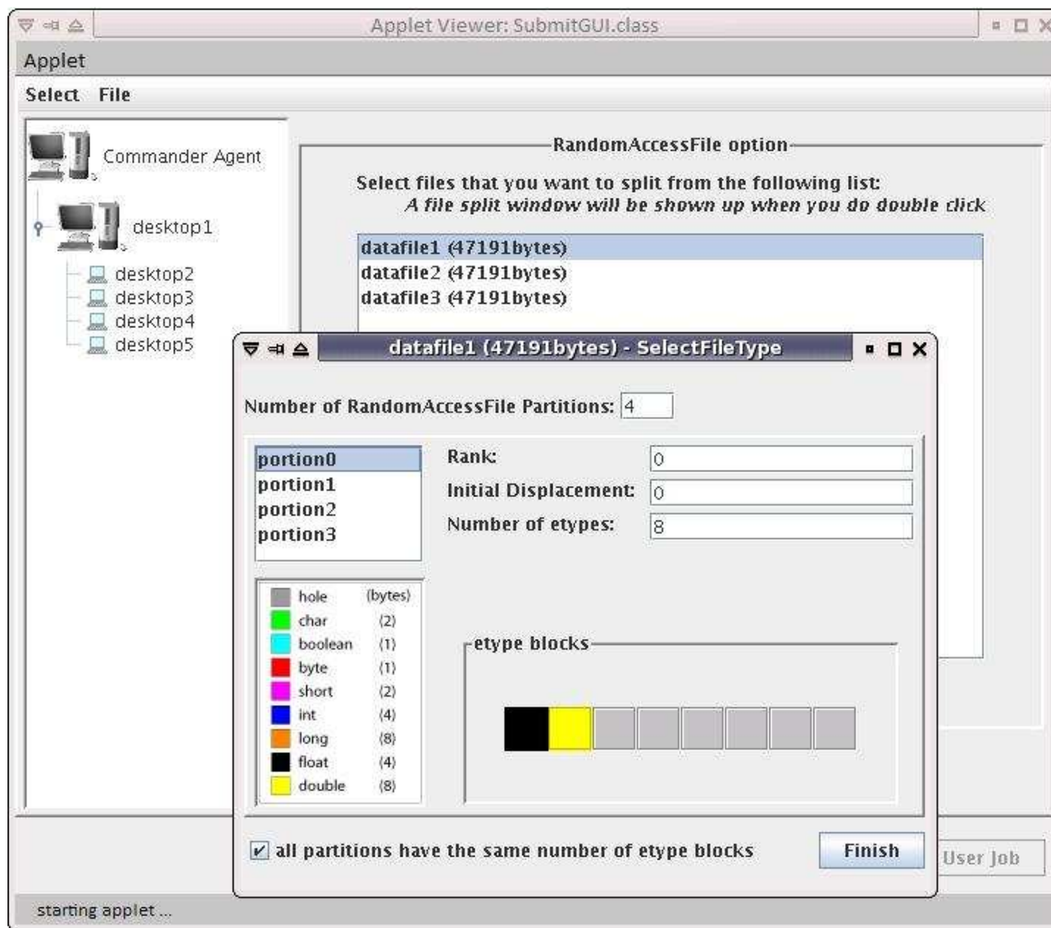

Figure 6. AgentTeamwork's GUI

whose size is four bytes, whereas the second specified as a *double*, starting from the $4th$ byte, tiled with one *etype* whose size takes eight bytes.

A challenge is how to physically partition a given file into stripes, each of which must correspond to a different rank's file view. A naïve 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. An improvement is to limit this file-scanning work to only once by reading an input file with a byte basis, checking which ranks need each byte, and writing it to the corresponding rank-based file stripes [11]. However, this scheme yet suffers from another drawback that cannot perform bulk read/write operations to a file, thus increasing OS interventions.

Rank: 0 ***RAFPartitionInfo***
Initial Displacement: 0
Total Bytes: 48
Barrier Status: WAITING
Collective Action: CLOSE

etype	displacement	#elements	etype size
float	**0th byte**	**1**	**4 bytes**
double	**4th byte**	**1**	**8 bytes**

Rank: 2 ***RAFPartitionInfo***
Initial Displacement: 0
Total Bytes: 48
Barrier Status: WAITING
Collective Action: CLOSE

etype	displacement	#elements	etype size
float	**24th byte**	**1**	**4 bytes**
double	**28th byte**	**1**	**8 bytes**

Rank: 1 ***RAFPartitionInfo***
Initial Displacement: 0
Total Bytes: 48
Barrier Status: WAITING
Collective Action: CLOSE

etype	displacement	#elements	etype size
float	**12th byte**	**1**	**4 bytes**
double	**16th byte**	**1**	**8 bytes**

Rank: 3 ***RAFPartitionInfo***
Initial Displacement: 0
Total Bytes: 48
Barrier Status: WAITING
Collective Action: CLOSE

etype	displacement	#elements	etype size
float	**36th byte**	**1**	**4 bytes**
double	**40th byte**	**1**	**8 bytes**

Figure 7. RAFPartitionInfo structures

Our final solution to file partitioning is to combine into one group all RAFPartitionInfos with the same displacement and size, to read an input file every 16MB through Java's *FileChannel* class[1], and to write as many bytes as tiled with consecutive *etype*s to the corresponding rank-based file stripes. For instance, consider the four *filetype*s described in Figure 7, all with the same displacement and the same size. Each rank has a pair of *etype* entries that occupy 12 consecutive bytes in total, (i.e., one *float* and one *double*) but start from a different displacement within the *filetype*. Therefore, our algorithm can write 12 bytes at a time to each rank-based file stripe. We assume that most parallel applications would have each rank repeat accessing a file stripe with the same size and with the same interval. Our solution makes the best effective use of this file-accessing pattern.

SubmitGUI completes this file-partitioning work with adding a RAFPartitionInfo to the top of the corresponding rank-based file stripe. Launched by SubmitGUI, a commander agent then reads these file stripes as independent files or receives them from SubmitGUI through *ssh* tunneling if it is running remotely. Although (entire) sequential files are read similarly into the commander, they can be distinguished from files stripes, using their header. When relaying file messages through an agent hierarchy, each agent duplicates them only if they are portions of a sequential file but not a file stripe. The next section details this file-transfer mechanism.

[1] 16MB is our experimental block size that can optimize FileChannel's file-reading performance.

3.2 File Transfer

Whichever files are sequential or striped, AgentTeamwork transfers them based on four strategies such as (1) file distribution in a hierarchy, (2) file aggregation, (3) file fragmentation, and (4) file collection along multi-paths.

First, file distribution in a hierarchy takes advantage of parallelism inherent to AgentTeamwork's agent tree for file distribution. (Although other network topologies could be conceivable, the underlying inter-agent communication must use UWAgents' agent hierarchy, and therefore we have selected a tree so as to avoid any overhead incurred by topology conversion.) Our strategy relays a file from a commander to all sentinel agents through their tree as duplicating the file at each tree level if necessary[2]. This would mitigate repetitive disk accesses and file-copying operations at a client site in particular if remote processes need to read the same collection of data files.

Second, file aggregation aims at mitigating inter-agent communication overheads by aggregating in one message all files that should be delivered to descendant agents in the same sub-tree, which thus reduces the number of file messages. Especially when random-access files are partitioned into stripes, (each accessed by a different process), this aggregation would improve the system performance by sending in one message all file stripes that will be accessed by agents in the same sub-tree.

Third, file fragmentation limits the size of each aggregated file message. If a file message is large enough to dominate network links, it is fragmented into smaller messages with a system-defined size. This strategy can also avoid the prolonged delay of user process executions.

Figure 8 describes an example flow for sending user files to sentinel agents, each dispatched to a different remote node to run a user process. Using the UWAgents mobile-agent execution platform, AgentTeamwork deploys a job in a new agent hierarchy where a commander recursively spawns sentinel agents whose identifier (simplified as *id*) is calculated from their parent $id \times 4+$ *a sequential number* (1-based if the parent is the commander, otherwise 0-based). From its id, a sentinel agent can calculate an MPI rank to be assigned to its user process.

[2] If a file is a rank-based file stripe, it is not duplicated on its way to the corresponding rank.

Each agent repeats a sequence of file aggregation, fragmentation, and hierarchical transfer every time it receives a new file message from its parent. To be more specific, each file message includes an additional set of name attributes formatted in a quartet of *agentId*, *fileName*, *sequence* and *mpiRank*, where *agentId* is a destination sentinel's id; *fileName* is the original name of the user file; *sequence* indicates in bytes which portion of the file is carried in this message; and *mpiRank* is the MPI rank of a process to read this file message. To aggregate file messages into one, an agent stores in a Java hash table all these messages and their name attributes whose *agentId* belongs to the same child or its descendants. This grouping work can be achieved by repeatedly dividing each file message's *agentId* attribute by four until it reaches an immediate child's id. We limit the size of a new file message with this Java hash table size, so that a large message destined for the same child agent will be sent in multiple hash tables. Upon receiving a file message from its parent, an agent extracts all the hash table entries and sorts them in their name attributes so as to regroup them in their same destination, namely their same *agentId* attribute.

The example in Figure 8 considers that a user has two data files named *inputFile1* and *inputFile2*, the former shared among three user processes with rank 1, 2, and 7; and the latter between two processes with rank 1 and 7. It also assumes that both files are small enough to fit in one hash table. A commander agent (denoted as *cdr*) reads and passes them in a hash table to the first sentinel (denoted as *snt*) with id 2. Since all these files have an eight-divisible *agentId*, sentinel 2 simply passes the table to sentinel 8 that thereafter regroups the table entries into two new hash tables, one forwarded to sentinels 32 and 128 whereas the other passed through sentinels 33 and 132 all the way to sentinel 528.

File collection starts as soon as a user process writes data to files. It uses an agent hierarchy in similar to file distribution while its direction is reversed from each sentinel to the commander agent. From its nature, the closer to the commander a sentinel agent is located, the busier traffic of file messages it is exposed to.

To alleviate this traffic congestion at higher-level agents, we have implemented two file-collection paths in their hierarchy as shown in Figure 9. One is an ordinary path to transfer file messages from a child to its parent agent. The other is a bypass (drawn thick in Figure 9) from a child to its grandparent as skipping over its parent agent. More specifically, we permitted sentinel agents with rank 0, 1, 2, 3, and

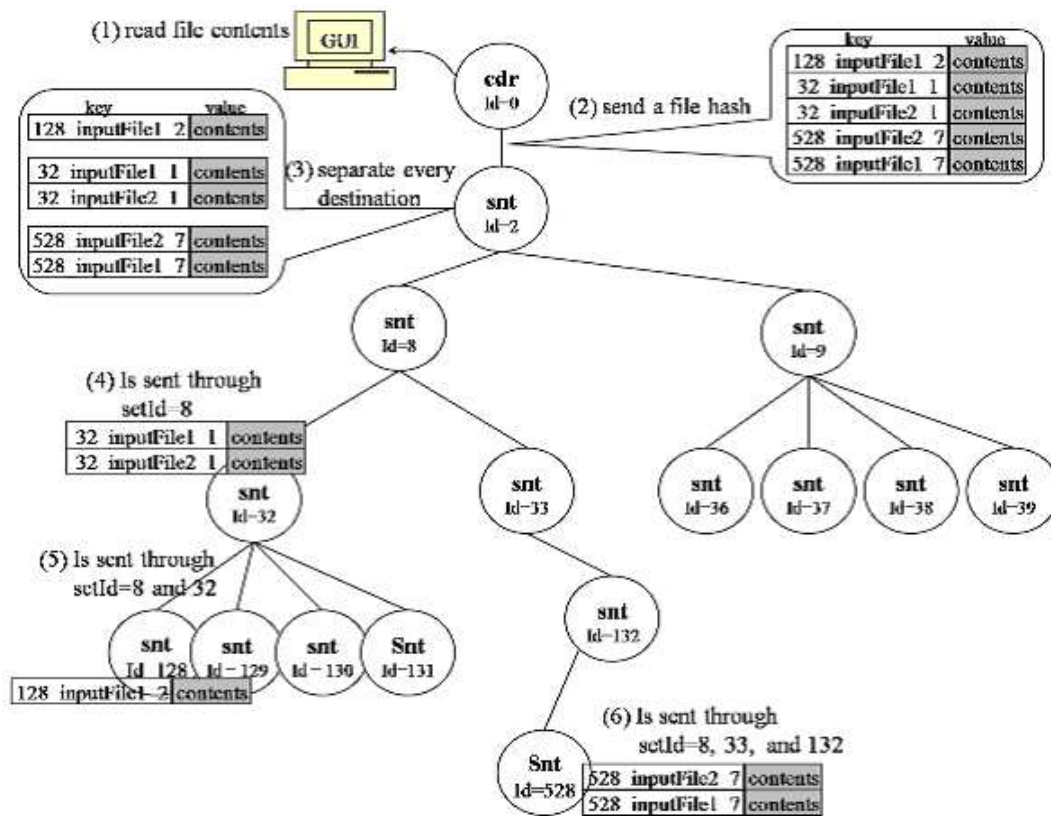

Figure 8. File distribution to sentinels in an agent hierarchy (for file read)

4 to send their file messages directly to the commander. We also created a bypass to an agent from its grandchild with a 4-divisible rank plus 1, (i.e., rank 5, 9, 13, 17, and 21 in Figure 9). This bypass allows each agent to receive file messages from four of its grandchildren ahead, thus relieves it from being stuck with accepting a bulk of messages from its children at once, furthermore alleviates the waste of memory space to spool in-transit messages, and therefore prevents unnecessary paging-out operations. In addition to file-collecting bypasses, each agent uses the conventional sliding-window algorithm to control its message flow to the parent or grandparent. The window size is currently adjusted between 2 and 15 messages.

3.3 File-Stripe Check-Pointing and Recovery

As briefed in Section 2, each sentinel agent instantiates a user program wrapper that periodically takes an execution snapshot of a given user program and resumes its execution from a crash at a new

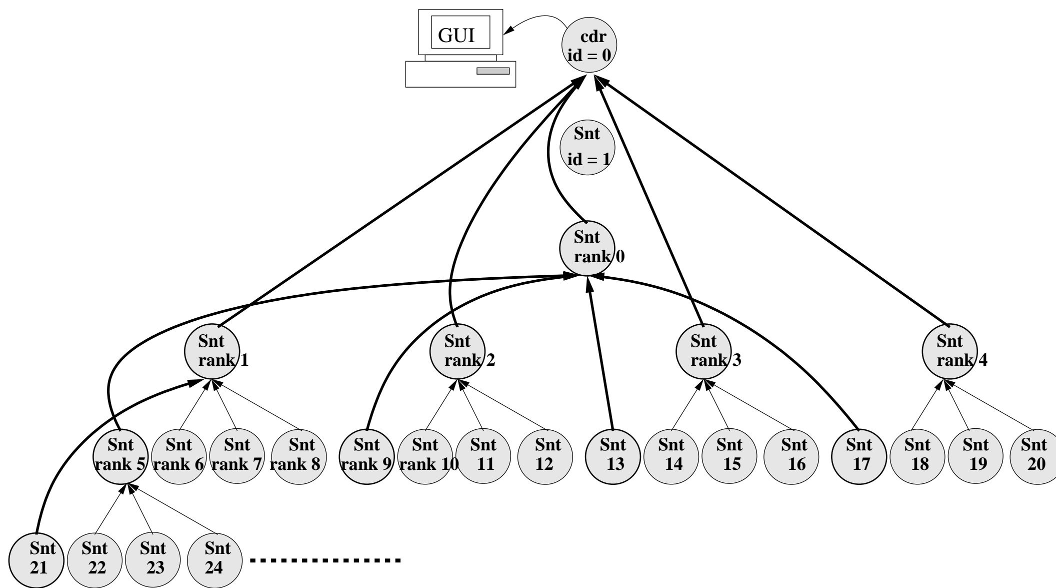

Figure 9. File collection to the commander in an agent hierarchy (for file write)

computing node. We perform this series of check-pointing and recovery work onto only file messages that a user program is currently accessing. In other words, since we assume that entire user files are always available from their original site, we only check-point file messages in transit through an agent hierarchy rather than replicate the whole files.

To serialize and check-point file messages with a user program, the wrapper creates and captures in a snapshot a GridFile object that buffers file messages to be read and written by the corresponding user program. Figure 10 shows file-message maintenance with GridFile. In addition to the main thread that executes a user program, an sentinel agent spawns two child threads named *input* and *output* threads. These three threads take the following roles:

1. **Input thread**: assigns a Java vector queue to a new file, registers this queue with the file name in GridFile's hash table, and repeats receiving a new file message from its parent and storing it in the corresponding queue. (Needless to say, if this sentinel agent has children, the input thread passes incoming messages to its descendants.)

2. **Main thread**: retrieves a queue from GridFile, reads messages from it, and deletes the queue in response to a user program's file open, read, and close respectively. For file create and write

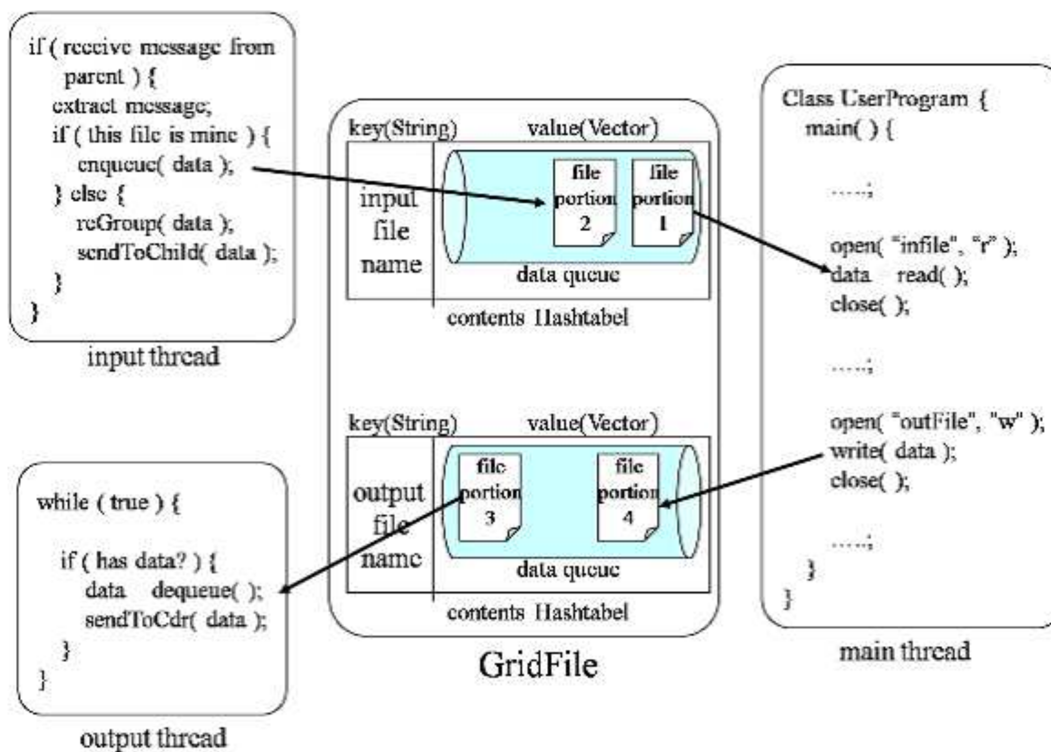

Figure 10. Files maintained by GridFile

operations, the main thread takes over the input thread's task, namely registering a new queue in GridFile and storing written data in the queue.

3. **Output thread**: takes charge of sending back user-written files to the commander agent. More specifically, it keeps checking GridFile's hash table to locate a user-registered queue, reading file messages from the queue, and sending them to the commander agent that then writes them back to a user-specified directory.

Of importance is to enforce mutual exclusion of GridFile's queues that are accessed from these three threads. Although the main thread can read from a file as many data as written by the input thread, the current implementation suspends the main thread to append data to an existing file until the file is completely filled by the input thread.

Another problem is how to handle a file whose total volume grows beyond memory space due to the mismatching speed in file read and write between the input/output and main threads. Our tentative solution provides a user with an option that temporarily stores files in each remote site's */tmp* directory,

in which case a sentinel agent, however, cannot resume those files with it upon a job migration.

Given the above check-pointing mechanism, we recover crashed file messages from their latest snapshot through the following three steps:

1. **Crash detection**: Since a parent and its child sentinels in their hierarchy exchange a ping and an acknowledgment periodically, a crash of a user program, (namely the corresponding sentinel agent) is detected by its parent or child. (A child detects a crash only when no action takes place during a period of (a ping interval $\times$ its agent *id*), which prevents multiple agents from detecting the same parent's crash simultaneously.)

2. **Snapshot retrieval**: The sentinel that has detected a crash sends a query to the corresponding bookkeeper agent, receives the crashed agent's latest snapshot, and resumes it at a new site. The crashed agent retrieves all file messages in its GridFile's queues as well as those in transit to its descendant agents. This in turn means that the descendants may receive duplicated file messages, in which case they can detect such duplication with each message's *sequence* attribute and simply discard the same messages.

3. **File-message resending**: Each sentinel must also take into account that it may not receive certain file messages from an agent crashed at a upper tree layer. This is because an agent may crash before check-pointing all its previous in-transit messages. To retrieve missing messages, a sentinel sends a query to the commander agent that then accesses the original user file and returns only missing file messages to the requester agent.

A similar scenario is applied to output files where their file messages are sent from each sentinel back to the commander and then written into its client user's local disk. However, unlike input files, intermediate agents do not use output files, and therefore their work is to simply discard duplicated file messages but not to request their descendants to resend messages.

3.4 File-Stripe Consistency Maintenance

Needless to say, Java does not incorporate the MPI-I/O concept in its file manipulations. If each rank accesses a different file stripe, it should use Java's RandomAccessFile class whose *seek*, *read*, and *write* methods allow each process to jump to and access its designated file stripe. We have reimplemented RandomAccessFile using AgentTeamwork's GridTcp and GridFile classes so as to facilitate data consistency and crash recovery. Our RandomAccessFile implementation is based on the following three strategies:

1. **Data allocation**: For each user process, the corresponding sentinel reconstructs a random-access file locally by tiling the actual stripes received from the commander and by zero-initializing those owned by the other processes.

2. **Read-caching but write-through accesses**: Each process can read and cache non-owning (and thus remote) stripes with a 512-byte basis for better performance. However, every write to a non-owning stripe flushes the corresponding local cache and sends new data immediately back to the remote process.

3. **Barrier synchronization**: RandomAccessFile automatically gets 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( )* enforce file-stripe consistency as follows. The read( ) function starts with scanning a collection of RAFPartitionInfos 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 or the file range exists in the local cache, 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 file stripe into a socket output stream while the local thread reads the data from a socket input stream into the user buffer. To reduce the number of read requests, the actual data transfer is carried out in a 512-byte block. The write( ) basically works in the similar sequence of those tasks except the following three details: (1) the data flow is reversed; (2) only a requested stripe is transfered; and (3) the corresponding local cache is flushed.

Despite write-through stripe accesses, it is still anticipated that two or more user processes can read different file portions from the same remote stripe, 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 stripe accesses, to flush cached stripes, and to thus allow all user processes to obtain identical stripe contents in the conventional weak consistency model. Its implementation is straightforward. 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 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. For this purpose, RAFPartitionInfo maintains the current barrier type and its progress in its *collective_action* and *barrier_status* variables respectively.

4 Performance Evaluation

We have measured AgentTeamwork's file distribution performance over two cluster systems, named *cluster-s* and *cluster-d*, which include single-core and dual-core computing nodes respectively as summarized in Table 1. The following subsections evaluate the performance of our four file-distribution techniques that have been discussed in Section 3: (1) file partitioning, (2) file transfer, (3) file check-pointing and recovery, and (4) consistency maintenance.

4.1 File Partitioning

Our experiments have covered the following three partitioning cases with a 256MB file that has been read every 16MB and partitioned into smaller stripes, each written back to the same directory where the

Cluster-s: a 32-node single-core cluster

Gateway node:		
	specification	**outbound**
	1.8GHz Xeon x2, 512MB memory, 70GB HD, and SunNFS 4.1.3 installed	100Mbps
Computing nodes:		
#nodes	**specification**	**inbound**
8	2.8GHz Xeon, 512MB memory, and 60GB HD	2Gbps
24	3.2GHz Xeon, 512MB memory, and 36GB HD	1Gbps

Cluster-d: a 32-node dual-core cluster

Gateway node:		
	specification	**outbound**
	1.5GHz Xeon, 512MB memory, 40GB HD, and SunNFS 4.1.3 installed	100Mbps
Computing nodes:		
#nodes	**specification**	**inbound**
16	2.13GHz Intel Core2, 1GB memory, and 40GB HD	100Mbps
16	1.8GHz Dual-CoreAMD Opteron, 1GB memory, and 40GB HD	1Gbps

Table 1. Cluster specifications

original file resides.

1. **Best case**: considers the entire file as one single *filetype* and partitions it by #processes into stripes, each consecutively tiled with *double*s and allocated to a different rank. Therefore, the *filetype* is visible to each rank as $256MB/\#processes$ contiguous *etype*s and $256MB/\#processes \times (\#processes - 1)$ *holes*. This best case mimics file access patterns as seen in distributed grep where each process searches for a given word by scanning a different contiguous file portion.

2. **Reasonable case**: first partitions the entire file by #processes into stripes and assumes each as a *filetype*. This *filetype* is further divided by #processes into smaller pieces, each then consecutively titled with *double*s and allocated to a different rank. In other words, the *filetype* is composed of a pair of contiguous *etype*s and *hole*s with their respective $256MB/(\#processes)^2$ and $256MB/(\#processes)^2 \times (\#processes - 1)$ size. This reasonable case abstracts mesh-based computation such as parallel matrix multiplication that may distribute matrix data from files to remote processes lined up in a mesh.

3. **Worst case**: assumes $double \times \#processes$ as a *filetype* and allocates each *double* of it to a different rank. To be more specific, each *filetype* is a pattern of an 8-byte *etype* and $8(\#processes - 1)$-byte holes where 8 bytes are the size of Java's *double*. While *etype* could even take a smaller data type such as *char* or *byte*, this *double*-based test case would be pathetic enough to burden each process with frequent file seeks and reads.

To compare with AgentTeamwork, we have also measured Java's performance for 256MB-file reads followed by writes to a different file. In particular for file reads, we distinguished two test cases: one through *BufferedInputStream* and the other through *FileChannel*. The former shows performance of typical file reads and writes, whereas the latter maps a file directly onto memory, performs fastest among all the test cases, and thus shows the bottom line of file read/write overheads.

Figure 11 compares performance among all those five test cases. Needless to say, the worse test case is the slowest, actually 10.7 times slower than FileChannel. On the other hand, the best and reasonable cases performed 2.6 times slower than FileChannel and surprisingly only 1.4 times slower than BufferedInputStream. To be even better, their growth rate has slowed down after 32 partitions, which demonstrates the processor scalability of AgentTeamwork's file-partitioning scheme for some applications including distributed grep and matrix multiplication.

4.2 File Transfer

We have analyzed AgentTeamwork's file transfer performance from three different viewpoints including: (1) hierarchical file duplication, (2) file fragmentation and pipelined transfer, and (3) random-access file distribution and collection.

4.2.1 Performance of Hierarchical File Duplication

File duplication overheads have been measured by having each of 64 computing nodes read the same file whose size varies from 8M to 256M bytes. We have compared the following three systems:

1. **AgentTeamwork**: Injected from the Unix shell, a commander agent accesses its local disk and reads a given file every 16MB into file messages. Those file messages are then forwarded, dupli-

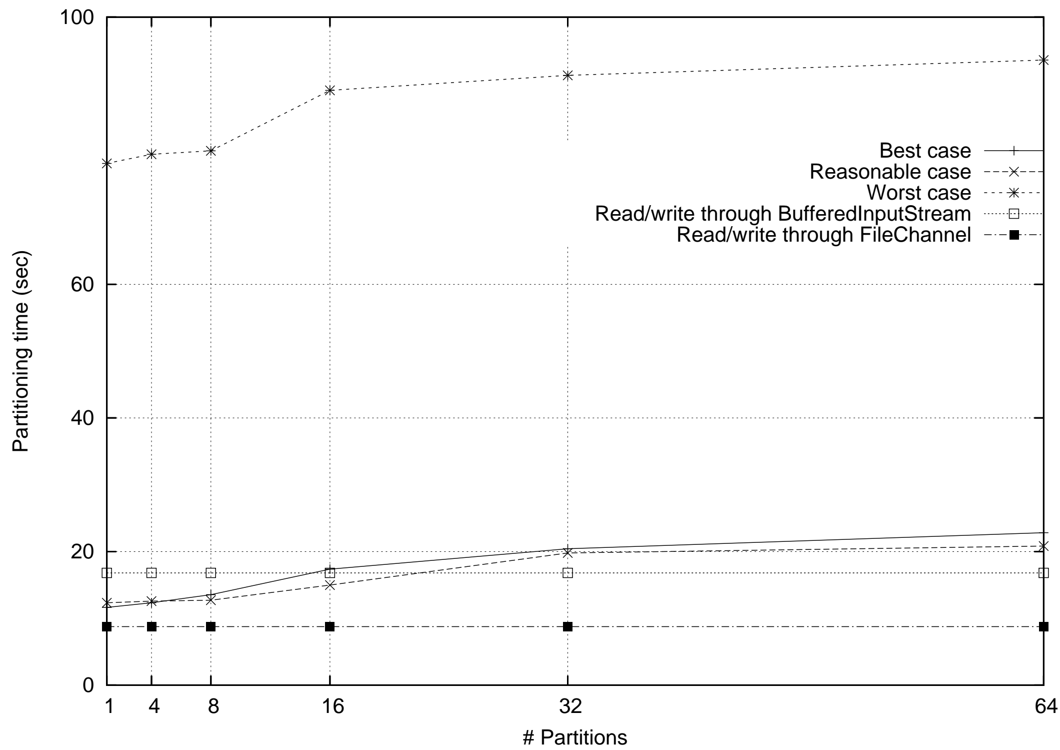

Figure 11. File-partitioning performance

cated, and delivered to 64 sentinels through their agent hierarchy. Upon receiving all messages, each sentinel acknowledges to the commander.

2. **Sun NFS**: We have coded the corresponding Java program that makes all 64 processes read the same file with a 16MB basis through SunNFS Version 4.1.3 and send a "completion" signal to their master process.

3. **User-Level Collective I/O**: To mitigate disk accesses at a user level, we have also revised the above NFS version using collective I/O [12] where the master process reads every 16MB block of a given file and immediately broadcasts it through Java sockets to all the slaves until reading up the file.

Figure 12 compares AgentTeamwork, Sun NFS, and user-level collective I/O. Despite that collective I/O was expected to drastically reduce the number of disk accesses at a user level, it actually performed only 0.8% to 4.1% faster than Sun NFS, (i.e., a non-collective I/O version). Two reasons are considered: (1) Sun NFS itself caches file data at a server side, (as known as server caching), which results in the

same effect as collective I/O, and (2) Sun NFS implements each read operation with an independent RPC, thus using a concurrent thread, whereas our collective I/O uses a single-threaded master process to broadcast file data to all the slaves, which may be occasionally synchronized with TCP's flow control.

On the other hand, AgentTeamwork ran faster than the others when broadcasting a 64MB or a larger file, while performing worst below 64M bytes. This is because, for a larger file, AgentTeamwork not only mitigates disk access overheads similarly to Sun NFS' server caching and collective I/O, but also distributes file duplication overheads across agents in their hierarchy. However, for a smaller file, each disk access and even each file duplication become negligible as compared to repetitive file relays through an agent hierarchy.

Additionally, we measured the performance of Sun NFS and user-level collective I/O that distributed a given file at once, thus with no repetitive 16MB-block transfer. For a 256MB file distribution, Sun NFS slightly improved its performance from 1478 to 1451 seconds, (i.e., 2.5% improvement), while collective I/O slowed down from 1464 to 1589 seconds (i.e., 8.5% speed-down). Therefore, 16MB-basis transfer did not benefit those two test cases.

4.2.2 Effect of File Fragmentation and Pipelined Transfer

We investigated the file-message size optimized to AgentTeamwork's file fragmentation and pipelined transfer as minimizing the size from 16M through 8M to 4M bytes. Similarly to hierarchical file duplication in Section 4.2.1, a commander agent reads a given file with a 16MB basis, however it immediately fragments each 16MB data into smaller file messages that are concurrently passed to a hierarchy of sentinel agents. Each sentinel multicasts an incoming file message to all its descendants, which in turn means that each message is duplicated at each tree level, relayed to its one-lower layer, and eventually delivered to all sentinels.

Figure 13 shows the performance of 16MB, 8MB, and 4MB fragmentation. The evaluation highlighted that, when transferring a 256MB file, 8MB fragmentation performed 1.16 times and 1.24 times faster than 16MB and 4MB fragmentation respectively. However, there was no difference between 8MB and 16MB fragmentation for smaller files than 256MB, while 4MB fragmentation performed slightly

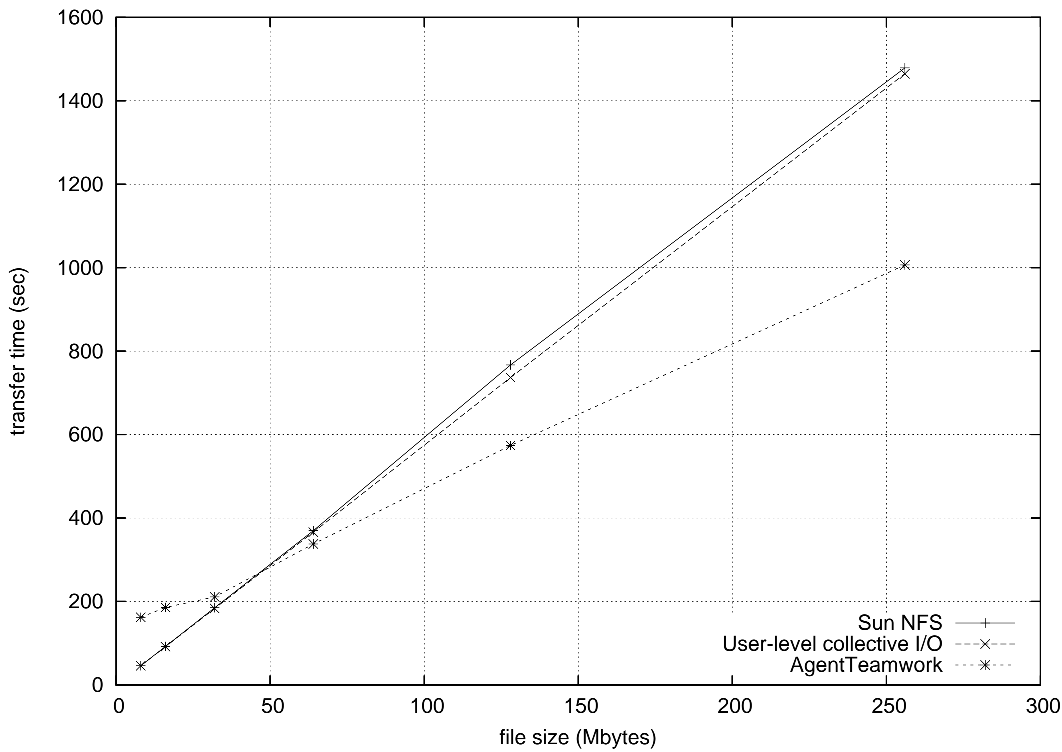

Figure 12. Performance comparison of file duplication

slower than the other two, (more specifically 1.02% to 1.19% slower than 8MB fragmentation).

Reasons for these results can be explained using two performance factors with regard to the fragment size: (1) the number of fragments passing through each agent and (2) the time of network links used to relay each fragment from one agent to another. The former grows as we decrease the fragment size to 4MB, which iterates more thread context switches within an agent to receive and relay each segment. On the other hand, the latter lasts longer as we increase the fragment size to 16MB, which synchronizes each agent more frequently with TCP's flow control. Under the current cluster configuration and Agent-Teamwork's software setting, 8MB fragmentation balances these two factors and therefore works best for larger file distribution.

However, we estimate that the fragment size could be enlarged to more than 8MB if we change the current conditions. First, since our cluster nodes are mixed up with 100Mbps, 1Gbps, and 2Gbps networks, we could at least upgrade 100Mbps links to 1Gbps, which will apparently reduce network latency and thus allow a larger fragment size without increasing the transfer time. Furthermore, shorter

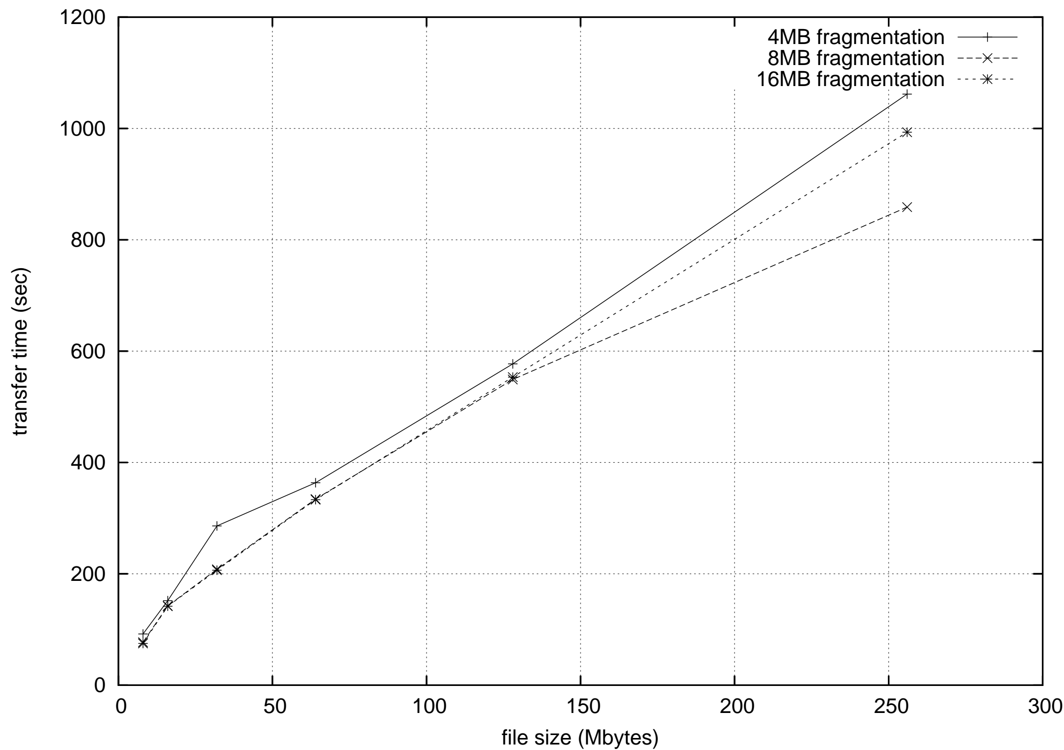

Figure 13. Effect of file fragmentation and pipelined transfer.

network latency could even mitigate each agent's overhead for periodical handshakes with its children, which permits each agent to spawn more agents, thus reduces the depth of agent hierarchy, but requires larger fragments so as to avoid frequent thread context switches.

4.2.3 Random-Access File Distribution and Collection

The performance of AgentTeamwork's random-access file transfer was measured in both file read and write operations, namely file distribution to and collection from remote processes.

The file-distribution experiment assumes that a random-access file has been partitioned into 64 file stripes *a priori*. We have then measured the total time elapsed for the entire sequence where 64 file stripes are read into a commander agent, relayed in parallel through an agent hierarchy, repeatedly aggregated or fragmented into 16MB file messages, and finally delivered to a different sentinel agent. (Note that this file-stripe transfer involves apparently no message duplication.)

On the other hand, the file-collection experiment first distributes a different file stripe to each of 64 sentinels, synchronizes all the sentinels, and starts a timer at the commander when receiving a signal

from the sentinel with rank 0. We have then measured the total time elapsed for the entire sequence where 64 file stripes are written by and sent from their respective sentinels up to the commander agent that stops the timer upon a completion.

Figure 14 demonstrates a substantial improvement by this parallel file-stripe distribution and collection. The file distribution performed 6.03 and 4.05 times faster than Sun NFS' and AgentTeamwork's entire file distributions respectively when a random-access file is 256MB long. Even added with 256MB file-partitioning overheads, (i.e., 20.8 and 93.5 seconds in the best and worst cases shown in Figure 11), AgentTeamwork's file-stripe distribution completes in a range between 265.9 and 338.6 seconds, which still yields 4.37 times to 5.56 times better performance than Sun NFS' entire file transfer. Obviously, the more user processes the more parallelism can be expected when distributing file stripes.

Notable in Figure 14 is that file-stripe collection performed even better than distribution, (more specifically 2.99 and 1.25 times better when handling a 16MB and 256MB random-access file respectively). We explain that this distinctive performance was resulted from two file-collection paths and the sliding window algorithm for flow control, both described in Section 3.2, which successfully avoided message congestion.

Our next concern is AgentTeamwork's performance for transferring larger files that may not be fitted to memory. For file distribution, as far as a user application keeps reading files delivered through an agent hierarchy, we estimate that AgentTeamwork could maintain its competitive performance. Our justification is that the system always partitions a file into small file messages with the maximum size of 16MB, each sent in pipeline, fitted in memory, and therefore causing no paging or disk accesses in its transit through intermediate nodes. However, if the final destinations do not read files immediately, file messages are accumulated up in GridFile's queues that eventually glow beyond memory space and thus induce page-out operations.

For file collection, as we increase the file size, AgentTeamwork would receive a more negative impact from message congestion that tends to be incurred on higher layers of its agent hierarchy. As a matter of fact, the performance superiority of file collection over distribution diminished from 2.99 to 1.25 times when increasing the file size from 16MB to 256MB. To be worse, if each process writes a different copy

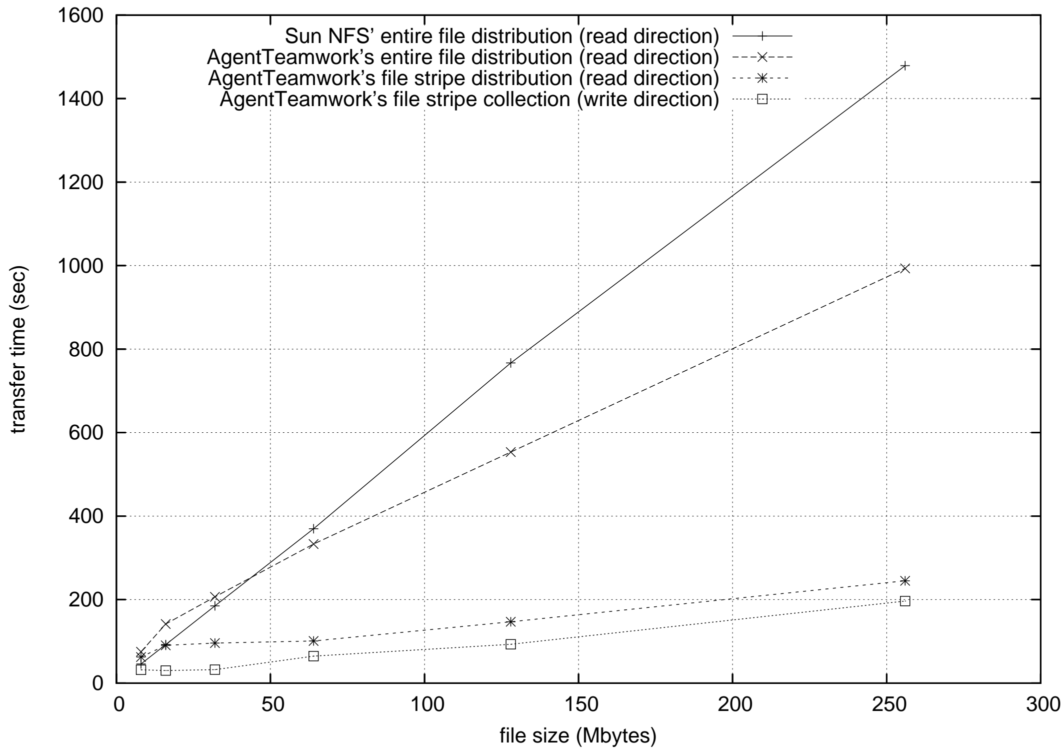

Figure 14. Performance of random-access file transfer

of an entire file rather than a file stripe, all copies must be collected at the commander, which significantly slows down the system performance. For instance, AgentTeamwork took 4,475.163 seconds to collect 64 copies of a 128MB file from sentinels. This is 8.09 times slower than the file duplication performance under the same conditions, (i.e., 128MB delivered to 64 sentinels.)

4.3 File-Stripe Check-Pointing and Recovery

We have assessed AgentTeamwork's file recovery overheads by distributing a 256MB file to 12 computing nodes with 8MB fragmentation and intentionally terminating a sentinel agent when it has received only the first two file messages, (i.e., the first 16MB)[3]. We considered three test cases: (1) killing no sentinel agent, (2) killing the rank-0 sentinel, and (3) killing the rank-11 sentinel. Test 1 simply distributes an entire 256MB file. Test 2 evaluates the largest overheads incurred from rank 0 to all the other sentinels. This is because rank 0 is located at the highest tree level among all sentinels. Test 3 evaluates the

[3]We are still working on AgentTeamwork's job recovery over multi-clusters (with 64 nodes in total).

Test cases	File distribution time
Test 1: no agent termination	262.60 seconds
Test 2: rank-0 termination	293.57 seconds
Test 3: rank-11 termination	266.68 seconds

Table 2. Agent recovery overheads

least overheads incurred by rank 11 at the lowest leaf, which thus affects no other agents.

Table 2 compares the file-transfer time of these three test cases. The results matched our estimation. Test 2 was approximately 31 seconds slower than test 1 whereas there was little difference between test 1 and test 3. Although this 31-second delay occupied only 11% of the entire file distribution, it will grow in linear to the number of sentinel agents, because each agent requests the commander agent to resend missing file messages. Therefore, we anticipate that, with 64 computing nodes, test 2's overheads would become $(31/11 \times 63) = 178$ seconds.

4.4 File-Stripe Consistency Maintenance

We have measured the performance of AgentTeamwork's RandomAccessFile class by exchanging *etype*s among different processes. Assuming N nodes involved in an experiment, all ranks are given the same *filetype* that includes N *etype*s, each tiled with four *double*s and owned by a different rank. We have coded two test programs named *master-workers* and *heartbeat* on AgentTeamwork. Master-workers makes rank 0 collect all *etype*s that the other ranks have modified, whereas heartbeat engages each rank in modifying its own *etype*s and thereafter reading its neighboring rank's *etype*s. These access patterns represent typical bag-of-task applications and spatial simulations.

To compare AgentTeamwork with the ordinary Java environment, we have also written the corresponding Java programs that use only Java-original classes (such as RandomAccessFile, Socket, and Thread) and run on Java Virtual Machine but not on AgentTeamwork. Executed over 2 to 16 computing nodes, they mimic AgentTeamwork's RandomAccessFile class by maintaining a file locally, spawning a thread for inter-process communication, and reading remote stripes through a socket.

Figure 15 shows time elapsed to run master-workers as increasing the file size up to 10MB as well

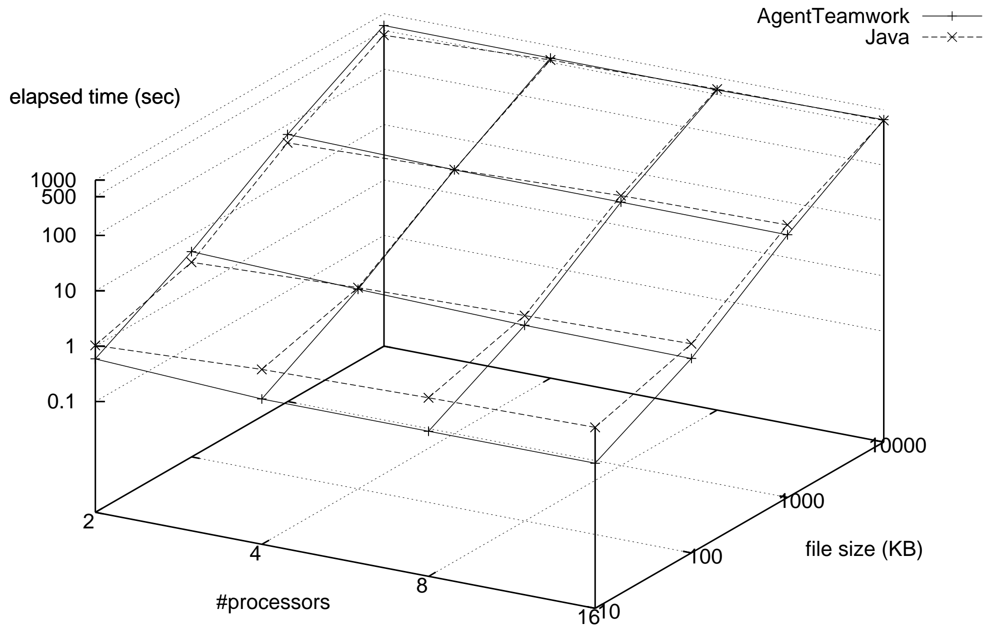

Figure 15. Performance of remote stripe accesses in master-workers

as the number of processors up to 16. Both AgentTeamwork and Java versions could not improve their performance with more computing nodes. This is because all modified stripes had to stem into rank 0, which thus could not obtain any benefit from parallelization. Although AgentTeamwork displayed its more overhead for stripe transfer as increasing the file size, its 512-byte stripe cache worked out better for small files, (more specifically a 10KB file in most cases).

Figure 16 compares heartbeat's performance between AgentTeamwork and the corresponding Java program. Contrary to master-workers, both AgentTeamwork and Java versions improved their performance as increasing the number of computing nodes. The file size has impacted to both versions in the similar manner to master-workers. The Java version performed 1.38 to 2.55 times better than AgentTeamwork on 10MB-file exchange, whereas AgentTeamwork utilized its file-stripe cache more effectively on small files whose size is up to 1MB in most cases.

In summary, the results have demonstrated that AgentTeamwork's RandomAccessFile can allow multiple processes to share file stripes if they are exchanged uniformly as seen in heartbeat.

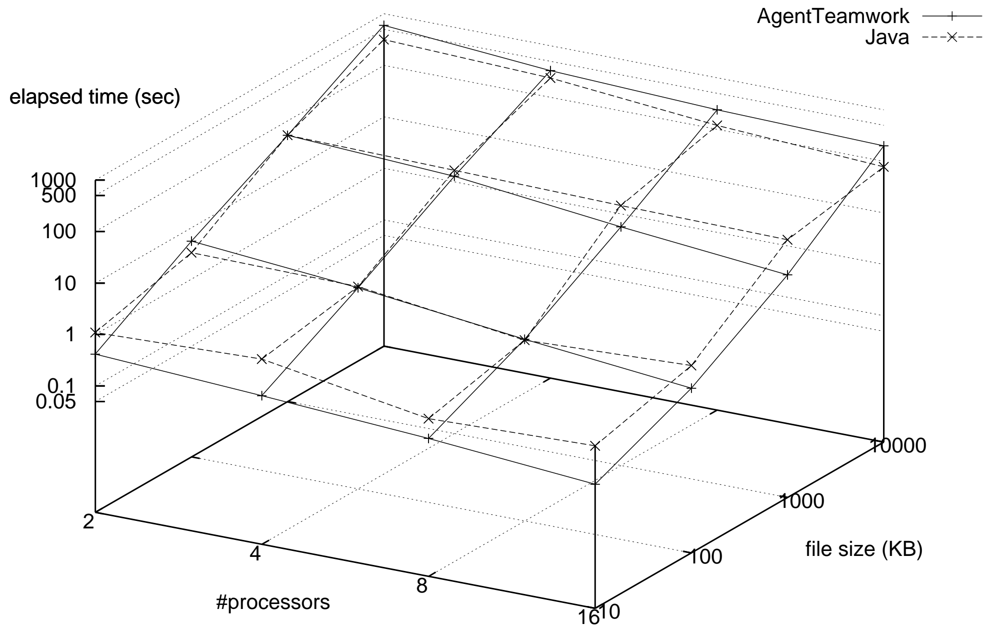

Figure 16. Performance of remote stripe accesses in heartbeat

5 Related Work

This section intends to clarify that AgentTeamwork's parallel file distribution differs from other grid systems in terms of (1) file caching and process invocation, (2) file partitioning and parallel transfer, (3) file duplication through a hierarchy, (4) data sieving and collective I/O, (5) replica management and file recovery, and (6) process check-pointing and recovery. In addition, as discussion item (7), we estimate the benefit of AgentTeamwork's file-distribution algorithm if it is applied to other grid-computing environments.

1. *File caching and process invocation*

   File caching at remote sites (where user processes are running) is a classical and typical technique to reduce both disk access and network traffic as seen in the most distributed and grid-computing systems. Legion provides a user with its low impact buffered interface that caches entire files in memory local to each user process [13]. Globus GASS facilitates open delegation where multiple

user processes can share a copy of the same file as far as they run on the same site [14]. Condor allows a user process to cache files in its local *tmp* disk as well as to access a remote file server on demand through Condor's remote system calls and I/O sockets [15]. Similarly, AgentTeamwork caches user files at each remote site. However, without waiting for entire files to be delivered, a sentinel agent starts a user program that can proceed its execution as far as file messages are made available in pipeline.

2. *File partitioning and parallel transfer*

   File partitioning permits multiple clients to retrieve a large file in parallel from a disk array or a collection of file servers. PVFS (Parallel Virtual File System) is such a system that has focused on parallel file access [16]. GridFTP uses multiple peer-to-peer channels to transfer a huge volume of data in parallel [17]. Parallel File Transfer Protocol has further accelerated cluster-to-cluster, more specifically PVFS-to-PVFS file transfer through a direct TCP connection between each pair of disk-dedicated cluster nodes [18]. Contrary to those systems, AgentTeamwork can neither take advantage of PVFS nor immediately transfer files that have not yet been partitioned. Although AgentTeamwork must read sequentially and thereafter partition a file through SubmitGUI, it is unique in transferring file strips through an agent hierarchy and re-aggregating them when relaying them to the same destination.

3. *File duplication through a hierarchy*

   File duplication through a hierarchy can distribute an identical file to multiple remote sites more effectively than one-to-one file transfer. FPFR (Fast Parallel File Replication) generates a spanning tree from a file server to multiple clients where intermediate tree nodes duplicate and relay a given file to their child nodes [19]. FPFR packetizes a file into smaller fragments, each of which may even take a different route for better performance. Using Globus RFT (Reliable File Transfer) [20], FPFR can detect faults in a tree and change its topology at run time. This work is quite close to ours in terms of the implementation concept, however AgentTeamwork can extend its agent hierarchy to multiple clusters and private network domains over gateways [8].

4. *Data sieving and collective I/O*

   Data sieving and collective I/O are intra- and inter-process I/O optimizations that mitigate disk accesses within a single and among multiple processes respectively by aggregating separate disk requests into one that can access a larger contiguous disk space. ROMIO is a well-known library that facilitates user-controllable data sieving and collective IO [12]. In particular, collective I/O can optimize disk accesses at different levels such as disks, servers, and clients, each termed disk-directed I/O [21], server-directed I/O [22], and two-phase I/O [23]. Since AgentTeamwork itself behaves as a client process from the file-system viewpoint, it can be categorized in two-phase I/O, where the first phase takes place in SubmitGUI's file-partitioning step and the second phase corresponds to file-stripe distribution in an agent hierarchy. A common problem is how to reduce communication overheads incurred by file distribution in the phase two. The inspector-executor collective I/O performs this task by exchanging file blocks among pairs of computing nodes as rearranging these pairs in a logarithmic order [24]. Although AgentTeamwork performs its file distribution in the same order, it differs in using a $log_x N$-based agent tree where x is the maximum number of children each agent can spawn and N is the total number of sentinel agents, (i.e., the number of user processes).

5. *Replica management and file recovery*

   File replication is a typical method to increase file-access availability, reduce file-access latency, and recover crashed files. For high file availability, Coda facilitates disconnected operations in a file system by allowing mobile clients to hoard files, emulate servers, and reintegrate files into their original servers [25]. For lower file-access latency, the Globus replica-management architecture has implemented an LDAP-based replica catalog API that allows users to select a given replica [26]. The EU DataGrid project has even automated this replica selection that locates the best replica with regard to network and storage access latencies [27]. For file recovery, Condor takes on-going execution snapshots and maintains them in a check-point server [15]. Globus RFT keeps track of the status of a peer-to-peer file transfer [20]. Both are targeting a single user process or the master process of master-worker-based applications that resumes its execution, in-

transit file data, and possibly worker processes. The Google file system takes a master-worker model to distribute a group of files to so-called *chunk servers* residing over multiple racks [28], so that stale computation can be repeated in parallel at different chunk servers for maintaining high-performance computation. Contrary to those systems, AgentTeamwork does not replicate an entire file. It places more emphasis on state-capturing of each remote process, assuming that the original file is always available at its user site. Each sentinel agent takes snapshots of its own user process including in-transit and on-memory file messages so as to keep pumping file messages to the process even after a crash.

6. *Process check-pointing and recovery*

   This feature is essential to keep executing long-run applications over a collection of computing nodes, some of which may encounter errors sooner or later. Of importance in parallel computing is how to restore broken communication among processes which may even lose in-transit messages. Condor-MW has focused on the master-worker model where the MW library in a master process keeps track of and retrieves connections to all its worker processes [29]. Legion allows its MPI applications to save their own data with MPI_FT_Save, restart all their execution from the top of the main function upon a crash, and restore the data from MPI_FT_Restore as directed from MPI_FT_Init [30]. SUMA/G has extended JVM and mpiJava functionality to repeat taking a process snapshot into its local disk automatically and to restore the latest global state of communicating processes as resuming in-transit messages [31]. Compared to these systems, AgentTeamwork does not restrict applications to the master-worker model for process check-pointing, resumes only a crashed process from MPI.Init( ), and distributes process snapshots to remote disks for more robustness. Its drawback is that AgentTeamwork's check-pointing mechanism is barrier-based among processes and user-initiated, (thus not automatic).

7. *Benefits to other grid-computing environments*

   Our tree-based parallel file distribution could benefit some grid-computing environments that are based on peer-to-peer file transfer or client-server-based network file systems. Condor [15],

Globus/GridFTP [17], and Nimrod/FT Server [32] have implemented their own mechanism to accelerate, parallelize, or secure peer-to-peer file transfer as focusing on their target applications. For instance, GridFTP maximizes file-transfer bandwidth between two end-to-end supercomputers. Yet in case if they were to distribute a file to a collection of computing nodes, our file-distribution algorithm could be used as an alternative in these environments. On the other hand, DUROC-based multi-cluster system [33] and Nimrod [34] use network file systems such as SunNFS or AFS to share files among computing nodes. As we have compared AgentTeamwork and SunNFS for their file distribution, (which thus facilitates file sharing) in Section 4.2, we feel that our algorithm could contribute better to those grid environments that use client-server-based network file systems.

6 Conclusions

Since AgentTeamwork allows mobile agents to migrate to and resume at a new remote site, its agent hierarchy can be considered not only as a self-remapping tree of user processes but also as dynamic file-distribution routes to the most available processor pool. Based on this viewpoint, we have implemented a sequence of file partitioning, distribution, collection, resumption, and consistency maintenance in an agent hierarchy:

1. AgentTeamwork's file-partitioning algorithm is based on the MPI-I/O concept and partitions a random-access file into each rank's file stripe by reading it every 16MB and tiling as many *etype*s as consecutively.

2. The file-distribution algorithm takes advantage of parallelism inherent to a hierarchy and speeds up its distribution performance with two strategies: (1) hierarchical and pipelined transfer, and (2) file fragmentation and aggregation at each tree level. The former enables each user process to start and advance its computation as much as possible. The latter sends files in blocks with the optimal size.

3. The file-collection algorithm has addressed output-file congestion on higher levels of an agent

hierarchy by relaying files over every other tree layer toward the commander agent.

4. The file-resumption algorithm allows each sentinel agent to take a repetitive snapshot of its user computation and to resume its parent and/or child with missing file messages when detecting a crash in its hierarchy. This resumption keeps providing a resumed process with necessary file messages.

5. The consistency-maintenance algorithm allows a user application to exchange their file stripes across multiple processes. We have implemented it using AgentTeamwork's GridTcp and GridFile as focusing on both file recovery and data-exchange performance.

Our measurements demonstrated the efficiency of these five file-transfer strategies. Needless to say, we must admit that network bandwidth and memory size would give a large impact onto performance results. For file distribution, faster network could allow an agent to spawn more children as well as relay larger file messages to each child, which would therefore reduce file-distribution time. For file collection, larger memory space enables an agent to hold more file messages in transit to the commander, mitigate the frequent flow control with its descendants, and thus relay output files faster to a user.

Finally, our next plan in AgentTeamwork is (1) to recover a large volume of file stripes that would overflow remote memory and thus must be saved in remote */tmp* disk; (2) to enhance SubmitGUI so as to monitor job migration, display file redirection, and provide users with help menus for MPI-I/O-based file partition; and (3) to allow each sentinel agent to invoke a C++ application that can call back AgentTeamwork's GridFile and RandomAccessFile for file manipulation. These new features would help AgentTeamwork facilitate a high-performance and fault-tolerant file-distribution environment to a wider range of users.

Acknowledgments

This research has been conducted under a three-year grant from National Science Foundation's Middleware Initiative (No.0438193). We are very grateful to our CSS graduates Joshua Phillips and Fumitaka Kawasaki for their programming contribution to an implementation of file-stripe consistency

maintenance and inter-cluster job resumption mechanisms on the AgentTeamwork system. We also would like to express our deep appreciation to Tim Rhoades, David Grimmer, and Meryll Larkin, the UWB Information Systems, for all their technical assistance in network and cluster management. Finally, we thank Chuck Jackels, Director of CSS Program at University of Washington, Bothell and Shinya Kobayashi, Chair of CS Department at Ehime University, for their support in the UWB-Ehime student-exchange program.

References

[1] Z. Balaton, G. Gombas, P. Kacsuk, A. Kornafeld, J. Kovacs, A. C. Morsi, G. Vida, N. Podhorszki, and T. Kiss, "SZTAKI Desktop Grid: a Modular and Scalable Way of Building Large Computing Grids," in *Proc. of Workshop on Large-Scale and Volatile Desktop Grids – PCGrid 2007 in conjuction with IEEE International Parallel and Distributed Processing Symposium*, (Long Beach, CA), pp. 26–30, IEEE, March 2007.

[2] M. Tachikawa, "PC Grid Computing – Using Increasingly Common and Powerful PCs to Supply Society with Ample Computing Resources," *Science & Technology Trends Quarterly Review*, vol. No.18, pp. 45–52, January 2006.

[3] M. Fukuda, K. Kashiwagi, and S. Kobayashi, "AgentTeamwork: Coordinating grid-computing jobs with mobile agents," *International Journal of Applied Intelligence*, vol. Vol.25, pp. 181–198, October 2006.

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

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

[6] 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), pp. 107–113, CSREA, June 2006.

[7] M. Fukuda, C. Ngo, E. Mak, and J. Morisaki, "Resource management and monitoring in AgentTeamwork grid computing middleware," in *Proc. of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing – PacRim'07*, (Victoria, BC), pp. 145–148, IEEE, August 2007.

[8] M. Fukuda, E. Horvath, and S. Lane, "Fault-tolerant job execution over multi-clusters using mobile agents," in *Proc. of the 2007 International Conference on Grid Computing and Applicaitons – CGA'07*, (Las Vegas, NV), pp. 123–129, CSREA, June 2007.

[9] mpiJava Home Page, "http://www.hpjava.org/mpijava.html," accessible as of February 2008.

[10] 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), pp. 259–262, IEEE, August 2005.

[11] J. Phillips, M. Fukuda, and J. Miyauchi, "A Java Implemenation of MPI-I/O-Oriented Random Acess File Class in AgentTeamwork Grid Computing Middleware," in *Proc. of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing – PacRim'07*, (Victoria, BC), pp. 149–152, IEEE, August 2007.

[12] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pp. 182–189, IEEE Computer Society Press, 1999.

[13] B. S. White, A. S. Grimshaw, and A. Nguyen-Tuong, "Grid-Based File Access: The Legion I/O Model," in *Proc. of the 9th IEEE International Symposium on High Performance Distributed Computing - HPDC'00*, (Pittsburgh, PA), pp. 165–174, IEEE CS, August 2000.

[14] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke, "GASS: a data movement and access service for wide area computing systems," in *Proc. of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, (Atlanta, GA), pp. 78–88, ACM Press, May 1999.

[15] Condor Team, "Conder version 6.6.11 manual, http://www.cs.wisc.edu/condor/manual/v6.6.11/, (accessible as of February 2008)," user manual, University of Wisconsin, Madison, WI, June 2006.

[16] Parallel Virtual File System, "http://www.pvfs.org/," accessible as of February 2008.

[17] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus striped GridFTP framework and server," in *Proc. of Super Computing 2005 - SC05*, (Seattle, WA), pp. 54–64, ACM Press, November 2005.

[18] D. Bhardwaj and R. Kumar, "A parallel file transfer protocol for clusters and gird systems," in *Proc. of the 1st International Conference on e-Science and Grid Computing*, (Melbourne, Australlia), pp. 248–254, IEEE CS, December 2005.

[19] R. Izmailov, S. Ganguly, and N. Tu, "Fast parallel file replication in data grid." in Homepage of GGF-10 Workshop: The Future of Grid Data Environments at http://ness.ac.uk/events/GGF10-DA/index.html (accessible as of February 2008), March 2004. Berlin, Germany.

[20] R. K. Madduri, C. S. Hood, and W. E. Allcock, "Reliable file transfer in grid environments," in *Proc. of the 27th Annual IEEE Conference on Local Computer Networks - LCN2002*, (Tampa, FL), pp. 737–738, IEEE-CS, November 2002.

[21] D. Kotz, "Disk-directed I/O for MIMD multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. Vol.15, pp. 41–74, February 1997.

[22] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-directed collective I/O in Panda," in *Proceedings of Supercomputing '95*, (San Diego, CA), pp. 57–60, IEEE CS, Decembe 1995.

[23] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy," in *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, (Newport Beach, CA), pp. 56–70, 1993.

[24] D. E. Singh, F. Isaila, J. C. Pichel, and J. Carretero, "A collective I/O implementation based on inspector-executor paradignm," in *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications – PDPTA 2007*, (Las Vegas, NV), pp. 683–689, CSREA, June 2007.

[25] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, (Pacific Grove, CA), pp. 213–225, ACM Press, October 1991.

[26] P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger, "File-based replicat management," *Future Generation Computer Systems*, vol. Vol.21, pp. 115–123, January 2005.

[27] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. Foster, "Secure, efficient data transport and replica management for high-performance data-intensive computing," in *Proceedings of the 18th IEEE Symposium on Mass Storage Systems and Technologies – MSS 2001*, (San Diego, CA), pp. 13–28, IEEE CS, April 2001.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 19th ACM Symposium on Operating Oystems Principles*, (Bolton Landing, NY), pp. 29–43, ACM Press, October 2003.

[29] Condor MW Homepage, "http://www.cs.wisc.edu/condor/mw/," accessible as of February 2007.

[30] A. Nguyen-Tuong, *Integrating Fault-Tolerance Techniques in Grid Applications*. PhD thesis, University of Virginia, Charlottesville, VA 22904, August 2000.

[31] Y. Cardinale, W. Pereira, and E. Hernnandez, "Extended mpiJava for distributed checkpointing and recovery," in *Proceedings of the 13th European PVMMPI Conference - LNCS 4192*, (Bonn, Germany), pp. 158–165, Springer, September 2006.

[32] D. Abramson, R. Sosic, J. Giddy, and B. Hall, "Nimrod: A tool for performing parametized simulations using distributed workstations," in *Proc. of the 4th IEEE International Symposium on High Performance Distributed Computing – HPDC-4*, (Pentagon City, VA), pp. 112–121, IEEE-CS, August 1995.

[33] K. Czajkowski, I. Foster, and C. Kesselman, "Resource co-allocation in computational grids," in *Proc. of the 8th IEEE Symposium on High Performance Distributed Computing – HPDC8*, (Redondo Beach, CA), pp. 219–228, August 1999.

[34] D. Abramson, J. Giddy, and L. Kotler, "High performance parametric modeling with nimrod/G: Killer application for the global grid?," in *Proc. of the 14th International Symposium on Parallel and Distributed Processing – ISPDP*, (Cancun, Mexico), pp. 520–528, IEEE-CS, May 2000.