An Implementation of Parallel File Distribution in an Agent Hierarchy [*]

Jumpei Miyauchi [1] Munehiro Fukuda[2†] Joshua Phillips [2]

[1]Computer Science, Ehime University, Matsuyama, Ehime 790-8577 Japan
[2]Computing & Software Systems, University of Washington, Bothell, WA 98011 USA

Abstract

AgentTeamwork coordinates parallel job execution in a hierarchy of mobile agents. A collection of specialized agents are deployed to remote sites including multiple clusters so as to launch, monitor, check-point, and resume a parallel and distributed-computing job. It is very important to develop an algorithm to deliver data files to each of remote processes in a timely fashion. By taking advantage of an agent hierarchy, we have implemented a file-distribution algorithm in AgentTeamwork that delivers user files through an agent tree by duplicating them at each tree level, dividing them into smaller partitions, and aggregating partitions in a larger fragment in transit to the same sub tree. This paper presents the details of AgentTeamwork's file-distribution algorithm and demonstrates its convincing performance.

Keywords: Parallel file transfer, grid middleware, job deployment, mobile agents

1 Introduction

One of the main motivations for grid computing is to allocate as many computing resources as requested to a resource-demanding application. Located remotely, those resources may be multiple clusters and even a collection of independent desktops, each not necessarily connected to the same network file system. This in turn means that efficient file transfer to remote sites is the key to remote job execution. Furthermore, it is not guaranteed that remote resources stay available as long as a given job is running. Therefore, we need a certain mechanism to redirect file transfer to a new site 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 [7]. 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 divides a file into stripes and delivers each one to a different process.

However, an entire file does not have to be passed to its destination at once. It can be fragmented and transferred in pipeline, which allows a user process to advance its computation as much as possible. At the same time, file stripes do not have to be delivered independently. They can be aggregated in one packet in transit to the same tree of descendant agents, which reduces network traffic. 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 partitions, aggregates partitions in a larger packet headed to the same descendant tree, and duplicates them if necessary.

To achieve fault-tolerant file delivery, each agent maintains in-coming file partitions 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 site, 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 stripes to the corresponding user process by resuming only lost stripes (rather than replicates an entire file as performed in the conventional replica management.)

This paper presents an implementation of file distribution in AgentTeamwork and demonstrates convincing performance. The rest of paper is organized as follows: Section 2 gives an overview of the AgentTeamwork system; Section 3 explains our file-distribution strategies; Section 4 presents the file-distribution performance; Section 6 differentiates our implementation from related work; and Section 5 concludes our discussions.

[*]This research is being conducted with full support from the National Science Foundation's Middleware Initiative (No.0438193).

[†]Corresponding author. Email: mfukuda@u.washington.edu, Phone: 1-425-352-3459, Fax: 1-425-352-5216

2 System Overview

AgentTeamwork is a grid-computing middleware system that coordinates parallel and fault-tolerant job execution with mobile agents [7]. A new computing node can join the system by running a UWAgents mobile-agent execution daemon to exchange agents with others [8]. 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.

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 internally facilitates error-recoverable TCP and file libraries, each named *GridTcp* and *GridFile* respectively, which serialize and de-serialize an execution snapshot with in-transit messages and buffered file data. A user program can take advantage of these fault-tolerant features by inheriting the *AteamProg* class that has re-implemented Java socket, file and even mpiJava classes [11] with *GridTcp* and *GridFile*.

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 32-33) as well as retrieve their contents from the latest snapshot (lines 27-28). At any point of time in its computation (lines 12-23), the application can take an on-going execution snapshot that is serialized and sent to a bookkeeper agent automatically (line 20). As mentioned above, it can also use Java-supported files and mpiJava classes whose objects are captured in snapshots as well (lines 14 and 21).

```
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      ...;
14      raf = new RandomAccessFile(    // create a file
15                new File("infile"),  // input file
16                "rw" );              // mode
17      int data = raf.read( );        // read a byte of data
18      raf.close( );                  // close
19      ...;
20      ateam.takeSnapshot(phase);     // check-pointing
21      MPI.COMM_WORLD.Barrier( )      // an MPI function
22      ...;
23    }
24    public static void main( String[] args ) {
25      MyApplication program = null;
26      if ( ateam.isResumed( ) ) {    // program resumption
27        program = (MyApplication)
28           ateam.retrieveLocalVar( "program" );
29        program.userRecovery( );
30      } else {                       // program initialization
31        MPI.Init( args );            // javaMPI invoked
32        program = new MyApplication( );
33        ateam.registerLocalVar( "program", program );
34      }
35      program.compute( );            //now go to computation
36      MPI.Finalize( args );
37  } }
```

Figure 1. File operations in AgentTeamwork's application

3 File-Transfer Strategies

This section focuses on AgentTeamwork's file distribution that facilitates the following three features: (1) hierarchical, aggregated, and fragmented file transfer, (2) file-stripe maintenance in memory for check-pointing, and (3) random access files.

3.1 Hierarchical, Aggregated, and Fragmented Transfer

It is intuitively natural to utilize parallelism inherent to an agent hierarchy for the purpose of delivering files to remote processes. In other words, our first file-distribution strategy is to relay a user file from a commander to all sentinel agents through their hierarchy as duplicating the file at each tree level. This would mitigate repetitive disk accesses and file copying operations at a client site in particular if remote processes need to read an identical set of data files.

It is obviously performance-effective to reduce the number of file transfers from one to another agent, (and thus to alleviate inter-agent communication overhead). For this purpose, our second file-distribution strategy is to aggregate in one inter-agent message all files that should be delivered to descendant agents in the same subtree. Especially when random access files are partitioned in stripes, each accessed by a different process, this aggregation would improve system performance by sending in one message all file stripes that will be accessed by the same subtree of agents.

It is on the other hand crucial to limit the size of each aggregated-file message (simplified as a file message in the following discussions) in order to avoid not only the extended use of network links but also the prolonged delay of user process executions. Therefore, our third strategy is to fragment aggregated files into smaller messages with a system-defined size.

Figure 2 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.

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 or each file stripe includes an additional set of name attributes formatted in a triplet of *agentId*, *fileName*, and *mpiRank*, where *agentId* is a destination sentinel's id; *fileName* is the original name of a user file; and *mpiRank* is the MPI rank of a process to read this file (stripe). To create a file message, an agent stores in a Java hash table all files and their name attributes whose *agentId* belongs to the same child or its descendants. This grouping work can be achieved by dividing each file's *agentId* attribute by four repeatedly until it reaches an immediate child's id. We limit the size of each file message with this Java hash table size, so that a large collection of files 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 files from the hash table and sorts them in their name attributes so as to group them in their same destination, namely their same *agentId* attribute.

The example in Figure 2 considers that a user has two data files named *inputFile1* and *inputFile2*, the former shared among three user processes with rank 0-2 and the latter among two processes with rank 0-1. It also assumes that both files are small enough to fit to 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 re-groups these file partitions into two hash tables, one forwarded to sentinels 32 and 128 whereas the other passed through sentinels 33 and 132 all the way to sentinel 528.

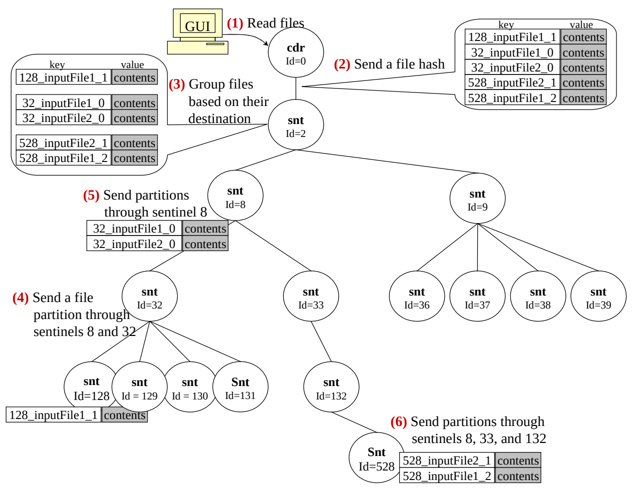

GUI
(1) Read files
cdr Id=0
(2) Send a file hash
key value
128_inputFile1_1 contents
32_inputFile1_0 contents
32_inputFile2_0 contents
528_inputFile2_1 contents
528_inputFile1_2 contents
(3) Group files based on their destination
key value
128_inputFile1_1 contents
32_inputFile1_0 contents
32_inputFile2_0 contents
528_inputFile2_1 contents
528_inputFile1_2 contents
snt Id=2
(5) Send partitions through sentinel 8
snt Id=8
32_inputFile1_0 contents
32_inputFile2_0 contents
snt Id=9
(4) Send a file partition through sentinels 8 and 32
snt Id=32
snt Id=33
snt Id=36
snt Id=37
snt Id=38
snt Id=39
snt Id=128
snt Id = 129
snt Id = 130
Snt Id=131
128_inputFile1_1 contents
snt Id=132
(6) Send partitions through sentinels 8, 33, and 132
Snt Id=528
528_inputFile2_1 contents
528_inputFile1_2 contents

Figure 2. File transfer in an agent hierarchy

3.2 File-Stripe Maintenance in Memory For Check-Pointing

As briefed in Section 2, each sentinel agent creates a user program wrapper that periodically takes an execution snapshot of a given user program. To serialize file data with the user program, the wrapper creates and captures in a snapshot an GridFile object that buffers file data to be read and written by the corresponding user program.

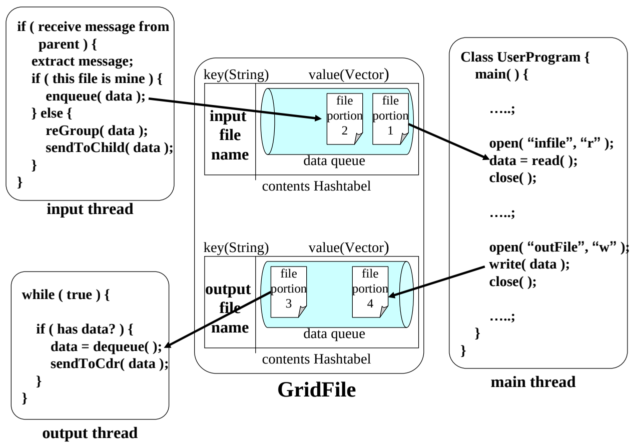
Figure 3 shows file-stripe maintenance with GridFile. In addition to the main thread that executes a user program, an sentinel agent spawns two child threads named *input* and *output* threads.

The input thread repeatedly receives a new file message from its parent sentinel, extracts file partitions from the message, assigns a Java vector queue to a new file, registers this queue with the file name in GridFile's hash table, and stores incoming file partitions in the corresponding queue. (Needless to say, if this sentinel agent has children, the input thread passes them a hash table of file partitions.)

The main thread retrieves a queue from GridFile, read data from it, and deletes it in response to a user program's file open, read, and close operations. For file create and write 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.

The 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 new user-written queue, reading file partitions from the queue, and sending them directly to the commander agent that then writes them back to a user-specified directory.

Of importance is to enforce mutual exclusion of Grid-

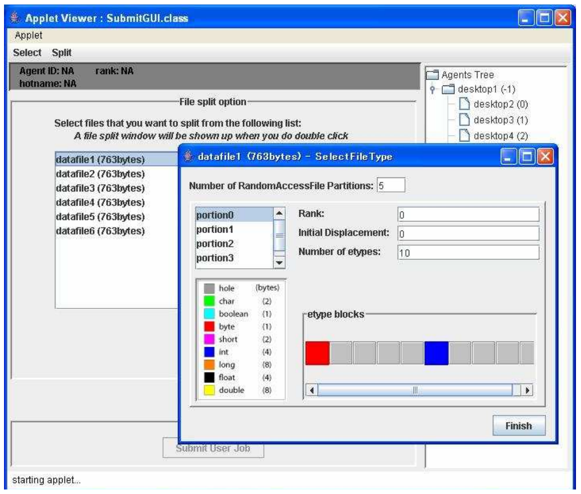


```
if ( receive message from
    parent ) {
  extract message;
  if ( this file is mine ) {
    enqueue( data );
  } else {
    reGroup( data );
    sendToChild( data );
  }
}
```

```
while ( true ) {

  if ( has data? ) {
    data = dequeue( );
    sendToCdr( data );
  }
}
```

```
Class UserProgram {
  main( ) {

    .....;

    open( "infile", "r" );
    data = read( );
    close( );

    .....;

    open( "outFile", "w" );
    write( data );
    close( );

    .....;
  }
}
```

Figure 3. Files maintained by GridFile



Applet Viewer : SubmitGUI.class

Agent ID: NA rank: NA
hotname: NA

File split option

Select files that you want to split from the following list:
A file split window will be shown up when you do double click

datafile1 (763bytes)
datafile2 (763bytes)
datafile3 (763bytes)
datafile4 (763bytes)
datafile5 (763bytes)
datafile6 (763bytes)

Agents Tree
desktop1 (-1)
desktop2 (0)
desktop3 (1)
desktop4 (2)

datafile1 (763bytes) - SelectFileType

Number of RandomAccessFile Partitions: 5

portion0
portion1
portion2
portion3

Rank: 0
Initial Displacement: 0
Number of etypes: 10

		(bytes)
▨	hole	
🟩	char	(2)
🟦	boolean	(1)
🟥	byte	(1)
🟪	short	(2)
🟦	int	(4)
🟧	long	(8)
⬛	float	(4)
🟨	double	(8)

starting applet...

Figure 4. File-stripe allocation

File'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 file partitions 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.

3.3 Random Access Files

AgentTeamwork makes random-access files viewed as Java's RandomAccessFile class to users. Moreover, we have incorporated the MPI-I/O's file concept [5] into this class so that a user can partition a given random-access file into stripes and allocate them to processes by specifying each rank's *filetype* and *etypes*. Figure 4 captures a snapshot of AgentTeamwork's GUI windows where a user has defined a *filetype* with ten different *etypes* and divided it into five stripes. Stripe 0 (or portion 0 in the figure) is composed of the first and the sixth *etypes*, and is allocated to rank 0.

Although an entire file is readable and writable to any process, AgentTeamwork delivers to each sentinel only the file stripes allocated to it, (while the file delivery follows the hierarchical transfer algorithm detailed in Section 3.1). Each user process can access those stripes in their original file position as if the entire file contents existed locally. If the process requests file data other than those allocated to it, the underlying RandomAccessFile object automatically establishes a GridTcp connection to the owner process that then transfers the necessary data to the requester.

Our current implementation does not enforce any strict ordering. Read and write operations are atomic and processed in a first-come first-serve basis. As an extension, we plan to add synchronization methods that mimic those defined in the MPI-IO standard.

4 Performance Evaluation

As described in Section 2, we have implemented the basic functionality of AgentTeamwork's specialized agents on top of the UWAgents mobile-agent execution platform. UWAgents facilitates agent mobility and inter-agent communication using Java RMI in its previous version and Java stream sockets in the current version, (each distinguished as AgentTeamwork/UWAgents-RMI and AgentTeamwork/UWAgents-Socket respectively in the following performance evaluation.)

The performance of AgentTeamwork's file distribution has been evaluated on a Giga Ethernet cluster of 24 DELL computing nodes, each with 3.2GHz Xeon CPU, 512MB memory and a 36GB SCSI hard disk. The following subsections compare AgentTeamwork with Sun NFS for their file transfer, examine the effect of AgentTeamwork's file fragmentation, and evaluate our implementation of random access files.

4.1 Comparison between AgentTeamwork and Sun NFS

We used a test case that allows each user process to access the same file whose size varies from 8M to 256M bytes. In AgentTeamwork, its file transfer time has been measured from a commander agent's injection to termination. This sequence includes (1) a commander reads a given file from its local disk; (2) the file is forwarded, duplicated, and delivered to 24 sentinels through the agent hierarchy; (3) each sentinel accepts the file; and (4) all agents acknowledge to the commander. In Sun NFS, we have coded the corresponding mpiJava program that makes all 24 ranks access the same file and send a "completion" signal to rank 0. In other words, we did not optimize this NFS version with well-known techniques such as two-phase I/O and data-sieving where rank 0 reads file contents and thereafter sends them in messages to the other ranks.

Figure 5 compares AgentTeamwork/UWAgents-RMI and Sun NFS for their file transfer speed. AgentTeamwork performed 1.7 times faster than Sun NFS when transferring a 256-MB file. Obviously, this large difference resulted from their file duplication schemes. AgentTeamwork accesses disk only one time and duplicates a file in each agent's memory space while relaying it through the agent hierarchy. On the other hand, Sun NFS accesses a file server in response to each remote user process, which makes 24 server accesses. However, for a smaller file with 2M through to 8M bytes, AgentTeamwork performed slower. This is because each disk access time is negligible as compared to repetitive file relays through an agent hierarchy.

Figure 5 also indicates that AgentTeamwork's file-transfer speed has slowed down when increasing the data size from 128M to 256M bytes. The main reason is that each agent must relay an entire file at once, allocate more memory to maintain the file, and postpone a launch of its user application until it completely receives the file. This is our motivation to packetize a file and to transfer file partitions in pipeline.

4.2 Effect of File Fragmentation and Pipelined Transfer

We have implemented AgentTeamwork's file fragmentation and pipelined transfer on top of UWAgents-Socket. Figure 6 compares AgentTeamwork/UWAgents-RMI's non-fragmented file transfer with AgentTeamwork/UWAgents-Socket's file fragmentation and pipelined transfer where each partition has been set to 1M, 2M, and 4M bytes.

The evaluation highlighted that AgentTeamwork's file fragmentation performed 1.9 times faster than non-fragmentation and 3.3 times better than Sun NFS when sending a 256-MB file. The results have also revealed that the partition size does not matter for a small file whereas the larger file the smaller partition works out to. In fact, 1-MB partitions performed 1.7 times faster than 4-MB partitions.

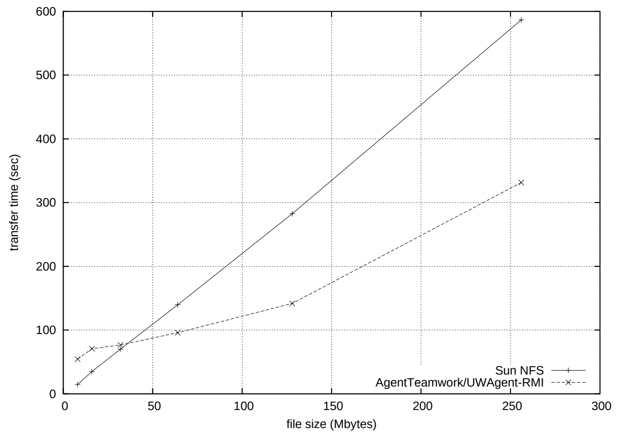


Figure 5. File transfer performance of AgentTeamwork and NFS.

4.3 Performance of Random-Access File Transfer

The proposed transfer of random-access files have been also implemented in AgentTeamwork/UWAgents-Socket. We have measured the time elapsed for the following sequence of random-access file transfer: (1) a commander agent reads 24 stripes of a given file, each to be delivered to a different sentinel; (2) the commander starts sending them in 1-MB partitions; (3) agents at each tree level relays file partitions as regrouping them or further dividing them; and (4) all agents send an acknowledgment to the commander when accepting their allocated file stripe.

Figure 7 compares this file-stripe transfer with an entire file transfer, both fragmented in 1-MB partitions [1]. The file-stripe transfer has yielded 1.35 and 4.5 times better performance than the entire file transfer and Sun NFS respectively when sending a 256-MB random access file. Although a commander agent still needs to send 256 messages, (each with a 1-MB file partition) as in sending an entire file, each agent at the bottom of an hierarchy receives only 11 messages. Obviously, the more user processes the better this transfer performs.

[1]We have used the same scale intentionally for the y-axis in Figures 5, 6, and 7 so as to compare them easily.

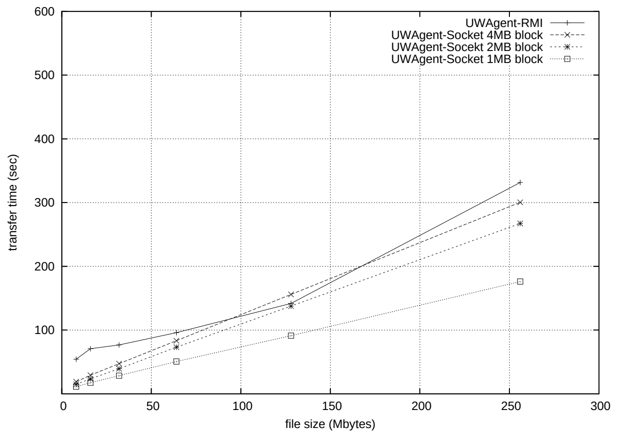


Figure 6. Effect of file fragmentation and pipelined transfer.

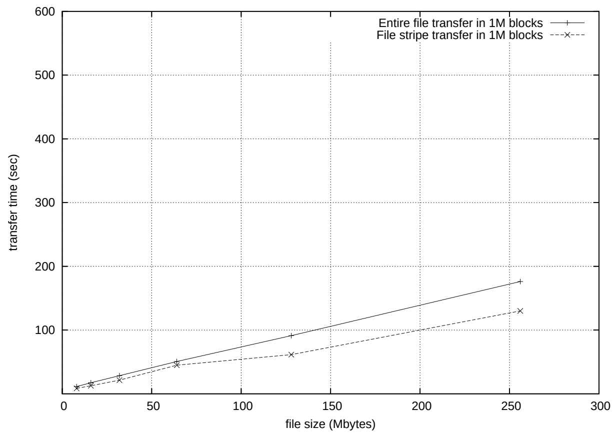


Figure 7. Performance of random-access file transfer

5 Related Work

In this section, we differentiate AgentTeamwork's parallel file distribution from its related work in terms of (1) remote file caching and process invocation, (2) file striping and parallel transfer, (3) file duplication through a hierarchy, and (4) remote file recovery.

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 the same file cache as far as they run on the same site [2]. 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 [4]. 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 contents are made available in pipeline.

File striping permits multiple clients to retrieve a large and shared file 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 [12]. GridFTP uses multiple peer-to-peer channels to transfer a huge volume of data in parallel [1]. 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 [3]. Contrary to those systems, AgentTeamwork cannot take advantage of PVFS nor immediately transfer files that have been already striped. Although AgentTeamwork must read sequentially and thereafter stripe a file through the commander agent, it is unique in transferring file strips through an agent hierarchy and re-aggregating them when relaying them to the same destination.

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 [9]. FPFR packetizes a file in smaller fragments, each of which may even take a different spanning tree for better performance. Using Globus RFT (Reliable File Transfer) [10], 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 [6].

Remove file recovery is necessary if a remote user process has modified files or is still in progress of file transfer. Condor takes an on-going execution snapshots and maintains them in a check-point server. Globus RFT keeps track of the status of a peer-to-peer file transfer. 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. On the other hand, AgentTeamwork places more emphasis on state-capturing of an entire parallel application where each sentinel takes snapshots of a different user process including in-transit file data and passes them to different bookkeepers.

6 Conclusions

We have implemented a file distribution algorithm using an agent hierarchy in the AgentTeamwork system. The strategies of our file distribution are three-fold: (1) hierarchical and pipelined transfer, (2) file striping and aggregation at each tree level, and (3) periodical file check-pointing. The first strategy enables each user process to start and advance its computation as much as possible. The second strategy sends files in blocks with the optimal size. Finally, the third strategy keeps providing a resumed process with necessary file data. The paper demonstrated the effectiveness of these file-distribution techniques.

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.

We understand that there is yet room for improvement in AgentTeamwork's file distribution. One is to recover a large volume of file stripes that would overflow remote memory and thus must be saved in remote */tmp* disk. The other is to maintain data consistency of files shared among different processes. With these new features, we feel that AgentTeamwork can facilitate a high-performance and fault-tolerant file-distribution environment.

Acknowledgments We are very grateful to Prof. Shinya Kobayashi, Chair of Computer Science Department at Ehime University for his support in human-resource allocation to our research.

References

[1] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The Globus striped gridftp framework and server. In *Proc. of Super Computing 2005 - SC05*, Seattle, WA, November 2005. ACM Press.

[2] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven 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*, pages 78–88, Atlanta, GA, May 1999. ACM Press.

[3] Dheeraj Bhardwaj and Rishi Kumar. A parallel file transfer protocol for clusters and gird systems. In *Proc. of the 1st International Conference on e-Science and Grid Computing*, pages 248–254, Melbourne, Australlia, December 2005. IEEE CS.

[4] Condor Team. Conder version 6.6.11 manual http://www.cs.wisc.edu/condor/manual/v6.6.11/. User manual, University of Wisconsin, Madison, WI, June 2006.

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

[6] Munehiro Fukuda. NSF SCI #0438193: Annual report for year 2006. Annual report, UW Bothell Distributed Systems Laboratory, Bothell , WA 98011, January 2007.

[7] Munehiro Fukuda, Koichi Kashiwagi, and Shinya Kobayashi. AgentTeamwork: Coordinating grid-computing jobs with mobile agents. *International Journal of Applied Intelligence*, Vol.25(No.2):181–198, October 2006.

[8] Munehiro Fukuda and Duncan Smith. UWAgents: A mobile agent system optimized for grid computing. In *Proc. of the 2006 International Conference on Grid Computing and Applicaitons – CGA'06*, pages 107–113, Las Vegas, NV, June 2006. CSREA.

[9] Rauf Izmailov, Samrat Ganguly, and Nan Tu. Fast parallel file replication in data grid. In *Proc. of Future of Grid Data Environments: A Global Grid Forum (GGF) Data Area Workshop*, Berlin, Germany, March 2004. GGF.

[10] 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*, pages 737–738, Tampa, FL, November 2002. IEEE-CS.

[11] mpiJava Home Page. http://www.hpjava.org/mpijava.html.

[12] Parallel Virtual File System. http://www.pvfs.org/.

[13] Brain S. White, Andrew S. Grimshaw, and Anh 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*, pages 165–174, Pittsburgh, PA, August 2000. IEEE CS.