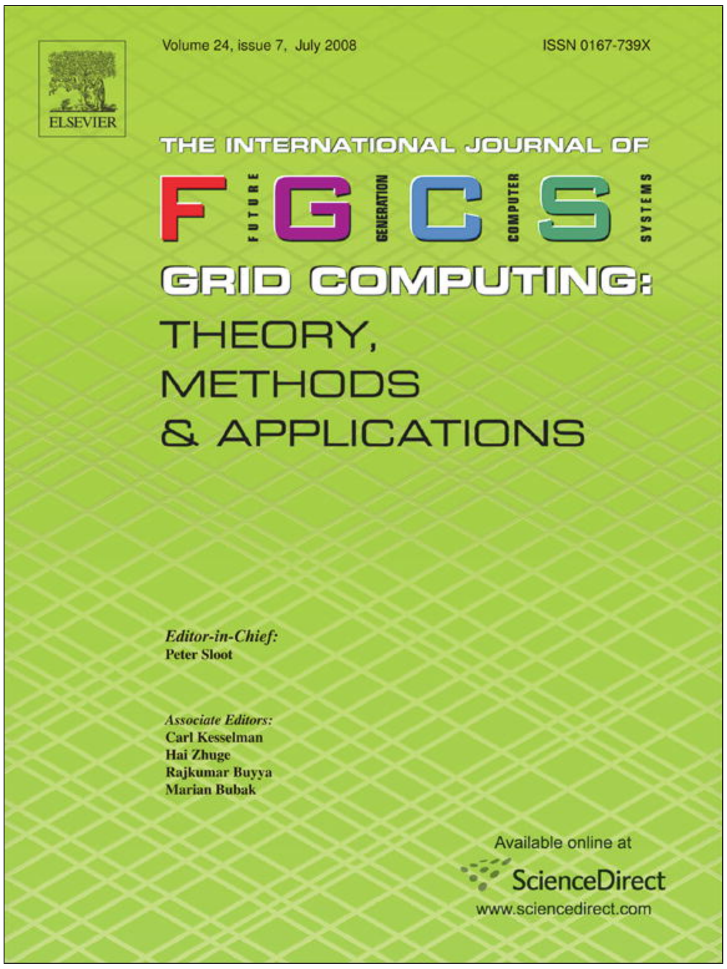
Volume 24, issue 7, July 2008
ISSN 0167-739X
ELSEVIER
THE INTERNATIONAL JOURNAL OF
FGCS
FUTURE GENERATION COMPUTER SYSTEMS
GRID COMPUTING:
THEORY, METHODS & APPLICATIONS
Editor-in-Chief:
Peter Sloot
Associate Editors:
Carl Kesselman
Hai Zhuge
Rajkumar Buyya
Marian Bubak
Available online at
ScienceDirect
www.sciencedirect.com

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Future Generation Computer Systems 24 (2008) 644–657
FGCS
FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Efficient reuse of replicated parallel data segments in computational grids

Sandip Tikar[a], Sathish Vadhiyar[b,*]

[a] EverGrid, Pune - 411045, India
[b] Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore - 560012, India

Received 28 April 2007; received in revised form 1 January 2008; accepted 3 January 2008
Available online 16 January 2008

Abstract

Grids are being used for executing parallel applications over remote resources. For executing a parallel application on a set of grid resources chosen by a user or a grid scheduler, the input data needed by the application is segmented according to the data distribution followed in the application and the data segments are distributed to the grid resources. The same input data may be used subsequently by different applications leading to multiple copies (replicas) of parallel data segments in various grid resources. The data needed for a parallel application can be gathered from the existing replicas onto the computational resources chosen by the grid scheduler for application execution. In this work, we have devised novel algorithms for determining "nearest" replica sites containing data segments needed by a parallel application executing on a set of resources with the objective of minimizing the time needed for transferring the data segments from the replica sites to the resources. We have tested our algorithms on different kinds of experimental setups. We find that the best algorithm varies according to the configuration of data servers and clients. In all cases, our algorithms performed better than the existing algorithms by at least 15%.
© 2008 Elsevier B.V. All rights reserved.

Keywords: Replica selection; Scheduling; Data movement; Grids; Parallel applications; Parallel Algorithms

1. Introduction

Computational grids have been found to be powerful research-beds for the execution of various kinds of parallel applications [2,4,19]. For the execution of a parallel application, a set of grid resources is chosen by a resource broker or grid scheduler [5]. The input data needed by the application is partitioned and distributed on the resources using a data distribution strategy [28].

In a virtual organization defined by a grid, different members of the organization or community may be interested in performing different parallel operations on the same input data. For example, applications involving parallel matrix operations have been executed on grids [19]. In these applications, different parts of a matrix are partitioned and distributed to the different resources used for a parallel matrix operation. The same matrix can be used as inputs to different matrix operations including LU factorization with multiple right-hand sides, multiplications with other right-hand sides, eigenvalue determination, calculation of norms etc. Different sets of resources can be selected by grid resource brokers for each of these operations. Similarly, large amounts of data will be generated in the high-energy physics experiments at CERN [9] and same portions of the data will be used for different kinds of processing by different users. Since the amount of data is large, some of the processing may involve parallel computations on the data whereby the data is partitioned and distributed among the resources used for parallel computations.

In grids, it is beneficial to have caching and replication policies so that data that is distributed on a set of resources for a parallel application is retained on those resources for access by subsequent parallel applications dealing with the same data. Thus, more replicas of input data will be created with different distributions on different sets of resources with more number of parallel computations on the same data. These replicas can help in reducing data access times for a computation and accordingly, various replica placement strategies have been proposed [21].

In this, paper we address the following challenge related to replica selection: *Given a set of replicas of an input data, with each replica corresponding to a partitioning and distribution of*

* Corresponding author.
E-mail address: vss@serc.iisc.ernet.in (S. Vadhiyar).

0167-739X/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2008.01.001

the input data on a set of resources (replica servers), and given a new set of resources (clients), on which a parallel application needing the same input data with a given distribution/partition will be executed, how to choose the "nearest" replica servers with the objective of minimizing the time for transferring the needed data segments from the replica servers to the client resources? Thus, for each data segment, an appropriate replica server has to be selected from many possible servers.

Previous efforts [3,10,20] have devised algorithms for selection of data servers for transferring data segments from multiple servers to a single client resource. In this paper, apart from extending two of the algorithms used in multiple server–single client scenario, we have also developed two new algorithms for selection of data servers and transferring data segments from multiple servers to multiple client resources for parallel application execution on the client resources. One algorithm considers the impact of simultaneous downloads on a data transfer and the other algorithm is based on collective download optimization used in parallel I/O.

In our work, we assume the existence of replica catalogs [6, 8] containing various information about replicas including locations of replica data segments and the data distribution schemes used for distributing the data to different replica resources. Based on the information contained in the catalog and information regarding a new parallel application, our algorithms determine the different sources/replica servers available for each of the data segments needed by the application. The algorithms then select a data source for each data segment needed by each process of the parallel application (client application) such that the time taken for fetching the entire set of parallel data by the parallel client for the parallel application is minimal. We evaluated the performance of our algorithms in terms of times needed for data transfers on various experimental setups that differed in client–server bandwidths and latencies. Based on our experiments, we conclude that different algorithms give best performances for different grid network settings. We also find that in all the experiments, our algorithms outperformed the multiple server–multiple client extensions of the existing algorithms by at least 15%.

In Section 2, our proposed algorithms for data selection and transfer are described in detail. In Section 3, different experimental settings in terms of different client–server bandwidths are considered and the performance of the algorithms on these settings are compared. In Section 4, related work is presented. Conclusions are given in Section 5 and future enhancements are listed in Section 6.

2. Algorithms for data selection

We have developed a total of 4 algorithms. Our algorithms assume that the user's data is distributed among the parallel resources with block-cyclic distribution corresponding to a block size. Thus different distributions of the same data correspond to different number of resources and/or different block sizes. We chose block-cyclic distribution since it is a popular data distribution strategy used by parallel applications [24]. The techniques developed in this work can easily be extended to other distributions. We use the terms *blocks* and *data segments* interchangeably to refer to the blocks and *block size* to refer to the size of the blocks in block-cyclic distributions.

In all our algorithms, the clients download blocks whose sizes are equal to *GCD_block_sizes*. *GCD_block_sizes* are obtained by calculating the GCD (Greatest Common Divisor) of the block sizes of the available data distributions in the replica servers and the data distribution used by the parallel client application. For example, if there are 4 replicas of parallel data corresponding to block sizes of 50, 150, 200 and 250 and the parallel client application uses block-cyclic distribution with block size of 100, our algorithms download data segments of size 50 from the replica servers. For most of our algorithms, we use latencies and bandwidths of the network links measured by periodic network probes.[1] The network probes execute every 2 minutes and write the latency and bandwidth values to probe files. These values are read from the probe files and used by our algorithms. Our implementations of the algorithms use the Globus GridFTP client [1] for transferring data segments from the servers to the clients. We do not use any specific optimizations for GridFTP data transfers including TCP buffer size tuning, parallel streaming etc.

The first two algorithms, *basic downloading* and $fastest_1$ are simple modifications of the algorithms developed for multiple servers-single client data transfers [3,7,10,20] to enable them for multiple servers–multiple clients downloads. These algorithms are used as base cases with which the performance of the other algorithms are compared. The algorithm that considers impact of simultaneous downloads, ISD, dynamically calculates the impact of simultaneous downloads of data segments from a set of servers to a set of clients on a given server-to-client data transfer. The last algorithm, *collective downloads*, is based on collective I/O optimization used in parallel I/O and is especially used in cases when the latencies of the links between the clients and the servers are high. Although the underlying principle has been used in parallel I/O, the algorithm incorporates significant decisions regarding selecting a replica out of many different replicas for data downloads. The following subsections describe each of the algorithms in detail.

2.1. Basic downloading algorithm

The *basic downloading algorithm* is the simplest of all the algorithms. It follows a workqueue model where a queue of pending blocks to be downloaded are maintained in each client. Each client opens multiple connections/threads to servers, one to each server that contains the blocks needed by the client. Each thread downloads a unique block from the server. When a thread finishes downloading a block, it starts downloading the next block contained in the corresponding server and that is not

[1] We are not using the popular grid tool, Network Weather Service (NWS) [32] for network measurements since currently, there are difficulties in executing NWS on Microgrid emulation tool on which we validate and compare our algorithms.

being downloaded by other threads. In this way, different clients download blocks needed by them simultaneously from different servers.

The basic downloading algorithm does not use network measurements for selecting replica servers for data downloads. The workqueue model followed in the basic downloading algorithm is a popular parallel programming model used in many situations especially when there are no dependencies between threads. Although, for our data download problem, there are not logical-level dependencies between data transfers by different threads, data downloads by one thread can impact data downloads by another thread thereby creating performance-level dependencies. Also, the basic download algorithm is considered to be unfriendly to networks since it can introduce network congestion due to unconstrained simultaneous downloads thereby impacting other network applications using the same network links. Although the basic downloading algorithm did not give good performance in the multiple server–single client scenario, we implemented the algorithm due to its simplicity and to verify if its performance improves in the multiple server–multiple client scenario.

2.2. *Fastest*$_1$

Fastest$_1$ is a promising algorithm and has been found to give the best performance for multiple server–single client data downloads [10]. In this algorithm, a client, for each block it wants to download, forms a list of servers containing the block. It chooses that server that minimizes $t * (l + 1)$ and downloads the block from the server. Here, t is the predicted time for downloading the block from the server when there is no contention. In our implementation, t is calculated at the beginning of the algorithm based on the size of the block, and the latency and bandwidth between the client and the server measured by our network probes. l refers to the number of current ongoing downloads from the server to the different clients.

The algorithm is implemented as a master–worker model. The master process is executed on one of the client machines that is "closest" to all other client machines in terms of the average of the bandwidths between the client and other clients. The worker/client processes, equal to the number of client machines, are executed on the client machines. The master process initially reads from network probe files and sends client–server latencies and bandwidths to all the clients. The master process also maintains *current_downloads* matrix whose (i, j) entry denotes the current number of data transfers between server j and client i. Whenever a worker/client process is ready to download, it sends a request to the master where the request is added to a request queue. The master processes the client request from the request queue and sends the *current_downloads* matrix to the client. The client chooses the server for downloading the block, spawns a process for downloading, and sends information regarding its current downloads from different servers to the master. The master uses this information to update the *current_downloads* matrix.

Algorithm 1 Fastest$_1$ Algorithm

```
1: $servers$: total number of servers; $clients$: total number of clients;
2: $lat, band$: $(clients \times servers)$ matrices denoting latencies and bandwidths, respectively, between clients and servers;
3: $curr\_dwlds$: $(clients \times servers)$ matrix where $curr\_dwlds[i, j]$ denotes the current number of data transfers between server j and client i;

4: Begin Master
5:     $request\_queue$: a queue of pending client requests;
6:     read probe files and fill $lat$ and $band$; $curr\_dwlds \leftarrow 0$;
7:     for $i \leftarrow 1, clients$ do
8:         send $lat, band$ to client i;
9:     end for
10:    while $request\_queue$ not empty do
11:        dequeue next request corresponding to client, $c$;
12:        send $curr\_dwlds$ to c; recv $curr\_dwlds[c]$ row from client c;
13:    end while
14: End Master

15: Begin Worker Client c
16:    $blocks(c)$: set of blocks needed by client, $c$;
17:    $serverswithblk(b)$: set of servers containing block, b;
18:    $blockList$: list of blocks remaining to be downloaded;
19:    $GCD\_block\_size$: GCD of block sizes of available and current data distributions;
20:    $min\_time$: minimum time to download a block;
21:    $min\_server$: server corresponding to $min\_time$;
22:    calculate $blocks$, $serverswithblk(b)$; $blockList \leftarrow blocks$;
23:    recv $lat, band$ from master;
24:    while $blockList$ is not empty do
25:        remove a block, b, from $blockList$;
26:        recv $curr\_dwlds$ from master;
27:        $min\_time \leftarrow Large$; $min\_server \leftarrow null$ ;
28:        for each server s $\in serverswithblk(b)$ do
29:            $t_{\text{orig}} = lat[c][s] + \frac{GCD\_block\_size}{band[c][s]}$ ;
30:            $total\_downloads = \sum_{i=1}^{clients} curr\_dwlds[i]\ [s]$;
31:            $predicted\_download\_time = t_{\text{orig}} \times (total\_downloads + 1)$ ;
32:            if $predicted\_download\_time < min\_time$ then
33:                $min\_time = predicted\_download\_time$; $min\_server = s$ ;
34:            end if
35:        end for
36:        spawn process to download b from $min\_server$;
37:        update $curr\_dwlds[c]$; send $curr\_dwlds[c]$ row to master;
38:    end while
39: End Worker Client c
```

The master terminates when there are no more requests in the request queue. The algorithm is shown in Algorithm 1.

Although *fastest*$_1$ adopts a simple strategy for server selection, it has been found to give good performance for multiple server–single client data downloads [10]. However, the following aspects of *fastest*$_1$ prevent the algorithm from being suitable for multiple server–multiple client data downloads. The major drawback of the *fastest*$_1$ algorithm is the scaling factor applied to the predicted time for downloading a block from a given server in the absence of contention, t_{orig}, to calculate the predicted time for downloading from the server in the presence of other data downloads. As can be seen in line 31 of the algorithm, the scaling factor is ($total\_downloads + 1$), which is the current number of downloads from the server to different clients added with 1 to indicate the potential download from the server to client c. Thus *fastest*$_1$ assumes a round-robin strategy with equal time quanta for scheduling multiple downloads from a given server. While round-robin is a popular strategy for scheduling CPU-intensive jobs, network-intensive data downloads are scheduled using complex system dynamics. Also, total_downloads is calculated as the plain sum of number of current downloads from the server to each client (line 30) thus giving equal weights of 1 to each client. Thus *fastest*$_1$ makes the assumption that the impact on a given download from a server to a client due to another download from the same server is the same irrespective of the location of the client involved in the other download. While this assumption is valid for multiple server–single client scenario where only one client is involved, it is unrealistic in multiple server–multiple clients downloads on a grid setting where different clients can be connected by same or different sets of network paths to a given server.

Also, the calculation of *predicted_download_time* for a given server, in line 31, is based on t_{orig}. By definition, t_{orig} is the time taken for downloading a block from the server to the client when there is no contention. Hence it should be calculated during initialization and before any of the data downloads. There are at least 2 approaches to calculate t_{orig}. One approach is to conduct an initial benchmarking procedure where downloads from the various servers with the GCD_block_size are performed and times taken for such downloads are measures as t_{orig} for the servers. Though this is a viable approach in the multiple server–single client scenario where the number of such downloads will be O (*servers*), it is not scalable for multiple servers–multiple clients downloads where the number of downloads will be O ($clients \times servers$). Moreover the benchmarking procedure will be redundant to the periodic network probes that are integral to many grid systems [32]. Another approach to calculate t_{orig} is the one followed in our algorithm in line 29, i.e., based on latencies and bandwidths measured by periodic network probes before data downloads. Although, this approach helps in determining the relative merits between different servers for a given client, it is not accurate since the message sizes used in network probes will be different from the block sizes used in the actual downloads [29]. Hence, the latencies and bandwidths calculated from network probes will not lead to accurate calculation of t_{orig}. Moreover, the calculation of t_{orig} before the downloads in both the approaches prevents *fastest*$_1$ from being adaptive to grid load dynamics and especially for large data downloads when external network loads can change during the downloads.

2.3. Algorithm based on impact of simultaneous downloads (ISD)

The ISD algorithm tries to remove/alleviate the problems mentioned for *fastest*$_1$ algorithm. In ISD, the predicted_download_time for downloading a block from a server, s, to a client, c, is calculated based on weighted sum of current number of downloads from the server to different clients instead of the plain sum used in *fastest*$_1$, and is given by:

$$t_{\text{orig}} \times \left(\left(\sum_{i \neq c} \text{weight}_i \times \text{current\_downloads}[i, s] \right) + \text{weight}_c \times (\text{current\_downloads}[c, s] + 1) \right). \quad (1)$$

The weights in the equation serve 2 purposes:

(1) The different weights for different clients take into account the different impacts of the downloads involving the clients and the server, s, on the potential download of data from the server to the client, c.
(2) Also, in ISD, t_{orig} is calculated based on latencies and bandwidths that are refreshed every 2 min by periodic network probes. Hence t_{orig}, the time to download a block of data from the server, s, to a client, c, in the absence of contention, is refreshed every 2 min thus making ISD more adaptive to external network load dynamics than *fastest*$_1$. But as discussed earlier, the use of latencies and bandwidths from network probes leads to prediction inaccuracies due to the different message sizes used for network probes and for data downloads. The weights in the equation also help to alleviate the prediction inaccuracies.

The working of ISD is shown in Algorithms 2–4.

The structure of the algorithm is very similar to *fastest*$_1$. The master process, through a separate thread, reads latencies and bandwidths from probe files every 2 min. When it processes a request from a client for the first time since the last refresh of latencies and bandwidths by network probes, it sends the network values to the client, c (lines 9–11 of Algorithm 2) and does not send the values to the same client until the next periodic refresh by the probes.

The client process then calculates *approx*_ t_{orig} for a given server, s, based on these latencies and bandwidths obtained from the master and the *GCD_block_size* (line 17 of Algorithm 3). At the time of obtaining the latencies and bandwidths, if there are no downloads from the server, s, to any client, *approx*_ t_{orig} is considered as t_{orig}, the time to download from s without contention (line 21 of Algorithm 3). If there are downloads from the server, s, i.e., there was contention, at the time of obtaining network parameters, then *approx*_ t_{orig} is scaled down by a factor to obtain t_{orig}, i.e. the time without contention (line 19 of Algorithm 3). The client then calculates the time to download

Algorithm 2 ISD Algorithm – Global Data Structures and Master

```
1: $servers$, $clients$, $lat$, $band$, $curr\_dwlds$: same as previous;

2: Begin  Master
3:     $request\_queue$: same as previous;
4:     spawn a thread that reads probe files every 2 minutes and updates lat and band ;
5:     $curr\_dwlds \leftarrow 0$;
6:     while $request\_queue$ not empty do
7:         dequeue next request corresponding to client, $c$ ;
8:         if c has sent download request for the first time since last read from probe files then
9:             send ($first\_time = 1$) to client c;
10:            $tempDownloads \leftarrow curr\_dwlds$ ;
11:            send $lat, band, tempDownloads$ to client c;
12:        else
13:            send ($first\_time = 0$) to client c;
14:        end if
15:        send $curr\_dwlds$ to c ; recv $curr\_dwlds[c]$ row from client c ;
16:    end while
17: End  Master
```

a block from the server using Eq. (1) (line 23 of Algorithm 3) and chooses the server for which the value of the equation, i.e. predicted time to download, is minimum for downloading the block.

The weights corresponding to different clients for different servers are maintained in a (clients × servers) matrix. The weights are initialized to 1 (line 6 of Algorithm 3), i.e., the algorithm starts with the same assumptions as *fastest*$_1$, and are then improved as the algorithm progresses. There are 2 kinds of weights: fixed and changeable. Fixed weights are those weights that do not change until the client obtains new latencies and bandwidths from the master while changeable weights change from the previous values. The (clients × servers) matrix, *fixed_weights*, denotes whether a given weight corresponding to a given client and server are fixed (value 1) or changeable (value 0). Initially, the *fixed_weights* matrix is initialized to 0 (line 6 of Algorithm 3).

After spawning a process for downloading the block from *min_server* (line 27 of Algorithm 3), the client *c* calculates the rate of progress of downloading after 2 seconds and determines the actual time to download, *actual_download_time*, based on the rate of progress, block size and the *latency[c,min_server]* (lines 2–4 of Algorithm 4). It then uses *actual_download_time* on the left-hand side of Eq. (1) to determine the weights (line 5 of Algorithm 4). Fixed weights are obtained when this equation is solved with only one unknown weight term on the right-hand side (line 8 of Algorithm 4). This happens when the rest of the terms either cancel when the corresponding number of downloads, *current_downloads[i, min_server]*, are 0 or when the corresponding weights, *weights[i, min_server]*, are fixed due to the solution of previous equations. Changeable weights are obtained when this equation has more than one

Algorithm 3 ISD Algorithm – Client

```
1: Begin  Worker Client c
2:     $blocks$, $serverswithblk(b)$, $blockList$, $GCD\_block\_size$, $min\_time$, $min\_server$: same as previous ;
3:     $weight$: a ($clients \times servers$) matrix containing weights between clients and servers used in Eqn. 1. ;
4:     $fixed\_weight$: a ($clients \times servers$) matrix. $weight[i, j]$ is fixed if $fixed\_weight[i, j] == 1$, and unchangeable otherwise.

5:     calculate $blocks, serverswithblk(b)$;
6:     $blockList \leftarrow blocks$; $weight \leftarrow 1$ ; $fixed\_weight \leftarrow 0$;
7:     while $blockList$ is not empty do
8:         remove a block, b, from $blockList$;
9:         recv $first\_time$ from master;
10:        if $first\_time == 1$ then
11:            recv $lat, band, tempDownloads$ from master;
12:            $fixed\_weight \leftarrow 0$;
13:        end if
14:        recv $curr\_dwlds$ from master;
15:        $min\_time \leftarrow Large$; $min\_server \leftarrow null$;
16:        for each server s $\in serverswithblk(b)$ do
17:            $approx\_t_{\rm orig} = lat[c][s] + \frac{GCD\_block\_size}{band[c][s]}$;
18:            if $tempDownloads[s] \neq 0$ then
19:                $t_{\rm orig} = \frac{approx\_t_{\rm orig}}{\sum_{i=1}^{clients} tempDownloads[i,s]}$ ;
20:            else
21:                $t_{\rm orig} = approx\_t_{\rm orig}$ ;
22:            end if
23:            $total\_downloads = \sum_{i \neq c}(weight[i][s] \times curr\_dwlds[i, s]) + weight[c][s] \times (curr\_dwlds[c, s] + 1)$;
24:            $predicted\_download\_time = t_{\rm orig} \times total\_downloads$;
25:            update $min\_time, min\_server$;
26:        end for
27:        spawn process to download b from $min\_server$ ;
28:        update $curr\_dwlds[c]$;
29:        send $curr\_dwlds[c]$ row to master;
30:        UpdateWeights();  ▹ a function to update weights ;
31:    end while
32: End  Worker Client c
```

unknown weight term on the right-hand side. In this case, after canceling the 0 terms and assigning fixed weights, the weights corresponding to the rest of the terms are assumed to be equal and solved.

The ISD algorithm is a “best-effort” algorithm where assumptions are made only when necessary. The problem of finding time to download from a server in the absence of any contention given the time to download from the server in the presence of a set of downloads from the server to a set of clients is a difficult problem to solve. Hence, we assume the round-robin scheduling strategy used in *fastest*$_1$ to determine this time in line 19 of Algorithm 3. Also, all the weights are initialized to 1 similar to *fastest*$_1$ and some weights will be changed according to Eq. (1) while

Algorithm 4 UpdateWeights()

```
1: procedure UPDATEWEIGHTS
2:     wait for 2 seconds ;
3:     find the amount downloaded from $min\_server$; calculate $actual\_band$ ;
4:     $actual\_download\_time = lat[c][min\_server] + \frac{GCD\_block\_size}{actual\_band}$ ;
5:     Solve actual_download_time = $t_{\text{orig}} \times (\sum_{i=1}^{clients}(weight[i][min\_server] \times curr\_dwlds[i, min\_server])$;
6:                                                                          ▷ In the above eqn., substitute all weights whose corresponding $fixed\_weight[][min\_server] == 1$. Cancel those terms whose corresponding $curr\_dwlds[][min\_server] == 0$. Treat weights of remaining unknown terms as equal. Solve the eqn. and update corresponding entries in $weight$ matrix.
7:     if number of unknown terms in eqn. == 1 and corresponds to a client c1 then
8:         $fixed\_weight[c1][min\_server] = 1$ ;
9:     end if
10: end procedure
```

some weights remain as 1. Thus, there will be an initial period of unfairness and unequal treatment of weights. Finally, when the equation has more than one unknown weight term during the weight update phase, we treat the corresponding weights as equal similar to *fastest*$_1$ in the absence of a better strategy. In spite of these small deficiencies, ISD is expected to perform better than *fastest*$_1$ for large data sizes on grid systems due to weight-based calculation of download impacts and its more adaptive capability to changing external load dynamics.

2.4. Collective download

Collective download is a traditional optimization in parallel I/O [27] where a set of parallel processors wants to read data that is distributed across a number of disks. The part of data that is required by one processor may not be available as a single contiguous chunk in a disk, but may be scattered among different disk locations and different disks. In *collective download*, each processor, instead of making multiple disk accesses to retrieve its part of data makes one single access to a disk and reads a contiguous chunk of data even if some parts of the chunk may belong to some other processors. The processors then exchange the required pieces of data between themselves. Thus the algorithm consists of a download/upload phase and a redistribution phase. The purpose of this algorithm is to decrease the cost due to multiple disk accesses and latencies in the download/upload phase at the expense of relatively much cheaper communication costs between the processors in the redistribution phase. This algorithm is useful in cases where there is a significant performance difference between the times needed for download/upload and for redistribution.

A similar technique can be used in our multiple servers–multiple clients data download scenario especially when the network characteristics between the clients and the servers are much worse when compared to the network characteristics between the clients themselves. This situation can happen when the clients and servers are located in different sites separated by high latency and low bandwidth WAN links. In our *collective download* algorithm, each client downloads a contiguous large data chunk from a set of servers in the download phase. The data chunk may be contained in a single server or may be contained in multiple servers as illustrated in Fig. 1(a) and (b). The client then sends the segments of the data chunk needed by other clients to the appropriate clients and receives the data segments needed by it from other clients in the redistribution phase. Our algorithm is designed in such a way that each client makes only one access to a server thereby incurring only one latency cost to the server while in the previous algorithms, a client could potentially make multiple accesses to a server.

Although collective download is widely used in parallel I/O optimization, our version of the algorithm for downloading in the presence of multiple replicas of parallel data is novel in the aspect of selecting a single parallel data replica for download. Our algorithm starts by finding the size of the data chunk to be downloaded by a single client as:

$$data\_chunk\_size = \frac{\text{total data size}}{\text{number of clients}}. \tag{2}$$

The algorithm then reads the latencies and bandwidths between the clients and servers from the probe files. For each possible server configuration, *svr_cfg*, i.e. a set of servers containing the parallel data, each client, *c*, finds the average time to download the data chunk from the servers in *svr_cfg*. This average time, *avg_dwld_time[c, svr_cfg]*, is calculated using *data_chunk_size*, and latency and bandwidth between the client and the servers in *svr_cfg*.

$$\begin{aligned} dwld\_time[c,s] &= lat[c,s] + \frac{data\_chunk\_size}{band[c,s]} \\ dwld\_sum[c, svr\_cfg] &= \sum_{s \in svr\_cfg} dwld\_time[c,s] \\ avg\_dwld\_time[c, svr\_cfg] &= \frac{dwld\_sum[c, svr\_cfg]}{count(svr\_config)} \end{aligned} \tag{3}$$

where count(*svr_cfg*) represents the number of servers in *svr_cfg*. The algorithm then finds the average of *avg_dwld_time[c,svr_cfg]*, *avg_avg_time[svr_cfg]*, over all the clients of the parallel application for the server configuration, *svr_cfg*.

$$\frac{\sum\limits_{c \in clients} avg\_dwld\_time[c, svr\_cfg]}{clients}. \tag{4}$$

The client then chooses that server configuration, out of many server configurations, for which *avg_avg_time[svr_cfg]* is minimum. This condition helps in minimizing the time required for the download phase. Data chunks from the servers in the chosen server configuration are downloaded by the clients in increasing order, i.e. the first data chunk from the first set of servers are downloaded by the first client, the second data chunk

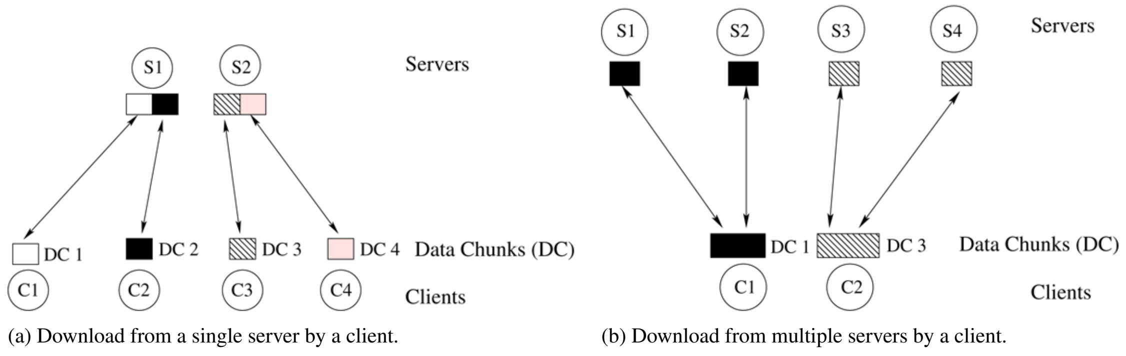

Fig. 1. Two cases in collective download.

from the second set of servers are downloaded by the second client and so on as shown in Fig. 1.

As mentioned earlier, after the downloading phase, some segments of the data chunk belong to the client while the other segments will be needed by other clients. The chosen server configuration can consist of the same parallel data in different distributions corresponding to different block sizes of the block-cyclic distributions, i.e., multiple replicas in the same set of servers. For each distribution or replica, the algorithm measures a metric called *avg_intersect*. For a given replica in the server configuration and for each client, the algorithm measures the percentage of data that will belong to the client out of the total amount of data that the client will download from a subset of servers in the server configuration. The average of this percentage across all clients is measured as *avg_intersect* for a given data distribution. The algorithm then chooses the data distribution or replica in a server configuration that maximizes the *avg_intersect* metric. The purpose of this maximization is that when most of the data that will be downloaded by the clients belong to the respective clients, only small amount of data will be exchanged among the clients in the redistribution phase thereby optimizing the redistribution phase. Thus our algorithm performs optimizations of both downloading and redistribution phases.

The algorithm is shown in Algorithm 5. Lines 6–21 perform optimization of download phase while lines 23–33 perform optimization of redistribution phase. Lines 36 and 37 are the download and redistribute phases, respectively, of the algorithm. As can be seen in Algorithm 5, unlike the ISD algorithm, the collective download algorithm is not adaptive to dynamic load changes during the data downloads since server selection decisions by the clients are based on the latencies and bandwidths collected before the data downloads. Also, when a server contains data chunks needed by more than one client as in Fig. 1(a), the collective download doesn't consider the impact of simultaneous downloads from the same server by the clients on the download performance. But in a grid setting involving slow WAN links, where slow latencies are the major bottlenecks, the advantage of using collective download due to single server access by a client can very well offset these disadvantages.

3. Experiments and results

We used 3 experiment configurations. The first setup is *Intra-Cluster Configuration* where the parallel servers and clients are the same machines in a cluster and thus inter-connected by local LAN network. This configuration is useful for scenarios when there is a powerful cluster in a grid and the grid scheduler repeatedly allocates the machines in this cluster for problem solving. Thus multiple replicas corresponding to multiple data distributions/block sizes are formed on the same sets of machines and new data distributions/replicas are formed on the same machines from the existing replicas using the algorithms discussed in the previous section. For experiments in this configuration, a cluster of 8 Intel Pentium 4 nodes connected by 100 Mbps Ethernet was used. Each node has a 2.8 GHz processor with 512 MB RAM, 80 GB hard disk and running Fedora Core 2.0 Linux 2.6.5 operating system. For all our experiments, we used network probes that use messages of sizes 64 KBytes to determine the periodic bandwidths of the links. The bandwidths observed on dedicated links of the 100 Mbps Intel cluster with the network probes are typically 87 Mbps. However, the bandwidths that were obtained on the dedicated links with the Globus GridFTP client [1] used in our algorithms were 6–38 Mbps for different message sizes. This is due to the additional overheads due to authentication and handshaking incurred by the Globus GridFTP protocol.

In the second setup, *Inter-Cluster Configuration*, the parallel data replicas are located in different clusters of different sites, with each replica entirely contained in a single cluster. The client machines, that execute the different parallel download algorithms to download data from the replica servers, are also located in a single cluster. The client and server clusters are typically separated by low bandwidth Internet links. This configuration is common when scheduling tightly-coupled parallel applications like ScaLAPACK [24] applications on grid systems. These applications were originally developed for tightly-coupled homogeneous systems and provide the highest performance when executed on machines within a cluster. Thus a grid scheduler can execute these applications at different clusters at different points of time, leading to formation of parallel data replicas in the clusters. At some point of time, when the parallel application is executed in a cluster, data

Algorithm 5 Collective Download Algorithm

```
1: $servers, clients, lat, band$: same as previous ;
2: $data\_chunk$: Part of the total data downloaded from a set of servers by a client ; $data\_chunk\_size$: Size of data chunk ;
3: $total\_svr\_cfgs$: number of distinct sets of servers containing replicas of parallel data ;
4:         ▷ If 3 replicas are contained in {svr-1, svr-2}, {svr-1, svr-3}, and {svr-2, svr-1} respectively, then $total\_svr\_cfgs$ is 2. i.e., only the first 2 sets are distinct.

5: read probe files and fill $lat$ and $band$ ; $data\_chunk\_size = \frac{total\_data\_size}{clients}$ ;
6: $min\_time \leftarrow Large$ ; $min\_svr\_cfg \leftarrow null$ ;
7: for each $svr\_cfg$ in $total\_svr\_cfgs$ do
8:     $total\_time[svr\_cfg] = 0$ ;
9:     for each client, $c$ in $clients$ do
10:        $count = 0$ ; $dwld\_sum[c, svr\_cfg] = 0$
11:        for each server, s, in $svr\_cfg$ do
12:            dwld_sum[c, svr_cfg] $+ =$ $(lat[c, s] + \frac{data\_chunk\_size}{band[c,s]})$; count ++;
13:        end for
14:        $avg\_dwld\_time[c, svr\_cfg] = \frac{dwld\_sum[c, svr\_cfg]}{count}$ ;
15:        $total\_time[svr\_cfg] + = avg\_dwld\_time\ [c, svr\_cfg]$ ;
16:    end for
17:    $avg\_avg\_time[svr\_cfg] = \frac{total\_time[svr\_cfg]}{clients}$ ;
18:    if $avg\_avg\_time[svr\_cfg] < min\_time$ then
19:        Reassign $min\_time$ and $min\_svr\_cfg$ ;
20:    end if
21: end for
22:         ▷ min_svr_cfg, a set of servers, can contain multiple replicas
23: $max\_intersect \leftarrow$ Small; max_rep $\leftarrow null$ ;
24: for each replica, $rep$, in $min\_svr\_cfg$ do
25:     $intersect\_amount = 0$ ;
26:     for each client, $c$ in $clients$ do
27:         $intersect\_amount + =$ percentage of $data\_chunk$ that will belong to c if downloaded from $rep$;
28:     end for
29:     $avg\_intersect = \frac{intersect\_amount}{clients}$ ;
30:     if $avg\_intersect > max\_intersect$ then
31:         $max\_intersect = avg\_intersect$ ; $max\_rep = rep$;
32:     end if
33: end for
34: parallel for
35: for each client, $c$, of parallel application do
36:     download c$^{th}$ $data\_chunk$ from c$^{th}$ sets of servers in $min\_svr\_cfg, max\_rep$ replica ;
37:     redistribute parts of $data\_chunk$ to & from other clients
38: end for
```

needed by the application can be obtained from the other clusters containing the replicas.

The final setup, *Chaotic Configuration* is where each of the replicas can be contained in machines from different clusters/sites. The client set of machines can also be from different clusters. This scenario happens when loosely-coupled applications like task farming applications are executed on grid systems. Because of the low communication complexities in these applications, a grid scheduler typically allocates many machines from different clusters for an application execution. Thus a single parallel replica may involve multiple clusters.

For the second and third experiment setup, we used an emulated grid environment. We used the Microgrid [16,25] emulation framework to validate, evaluate and compare our algorithms. For all our experiments, Microgrid was run on the cluster of 8 Intel Pentium 4 nodes. The use of Microgrid emulator for the second and third experiments can result in different runtimes when compared to experiments conducted on real machines. However, we show that emulation will not lead to significant changes in the relative differences in runtimes of the different algorithms. For this, we validate Microgrid by emulation of our local cluster. We conduct experiments corresponding to intra-cluster configuration on the emulated local cluster and compare the results with the corresponding results obtained on the real machines. Results on extensive validation of Microgrid for different environments were shown in earlier efforts [16,17].

The grid emulated using Microgrid was a virtual grid setting consisting of 6 sites: 1. University of California, San-Diego (UCSD), USA, 2. University of Urbana-Champaign (UIUC), USA, 3. University of Tennessee (UT), USA 4. University of Belfast, UK 5. Indian Institute of Science, India and 6. Kasetsart University, Thailand. The latencies and bandwidths of the links between the 6 sites were obtained offline by observing the average times for connections between the different sites and for downloading large files (Fedora Core binary downloads) from a site to the other sites. Table 1 shows the bandwidths and latencies between the different sites used in our Microgrid emulation. We used a total of 24 machines, 4 in each site, for our experiments. All the machines had equal CPU speeds. The bandwidths shown in Table 1 can fluctuate due to the presence of background traffic in wide-area networks. However, we do not emulate the background traffic in our experiments due to the lack of realistic wide-area traffic generators. Hence our results represent average behavior of our algorithms and can potentially overestimate performance. The times shown in the results represent the execution times of the algorithms and include both the data transfer times and algorithmic overheads.

3.1. Intra-cluster configuration

In this experimental setup, parallel data replicas, corresponding to a matrix of size 14 000 × 14 000 (1.56 GB), were formed in the 8-node Intel cluster. Each parallel replica corresponded to a number of processors and a block size for block-cyclic distribution of the data. The number of processors used were 2, 4 and 8 and the block sizes used were 50, 150, 200 and 250 leading to a total of 12 (3 × 4) combinations/replicas. The parallel client was run with different downloading algorithms on 8 machines of the cluster to download data from the replica servers

Table 1
Inter-site bandwidths (MB/s) and Latencies (s)

Sites	UCSD	UIUC	UT	UK	Th	India
UCSD	45.19, 0.09	4.16, 0.25	4.80, 0.28	1.96, 0.60	2.70, 0.27	0.77, 0.11
UIUC	4.27, 0.28	60.27, 0.06	2.07, 0.57	0.86, 1.35	2.07, 0.34	0.74, 0.10
UT	4.50, 0.28	2.02, 0.56	48.32, 0.09	2.88, 0.41	3.51, 0.33	0.15, 0.45
UK	2.83, 0.64	0.85, 1.35	2.86, 0.41	53.85, 0.06	0.62, 1.81	0.15, 0.29
Th	3.56, 0.29	3.68, 0.33	3.72, 0.32	0.62, 1.83	36.36, 0.05	0.15, 0.11
India	0.15, 0.12	5.81, 0.08	2.72, 0.44	4.36, 0.30	2.57, 0.13	41.41, 0.08

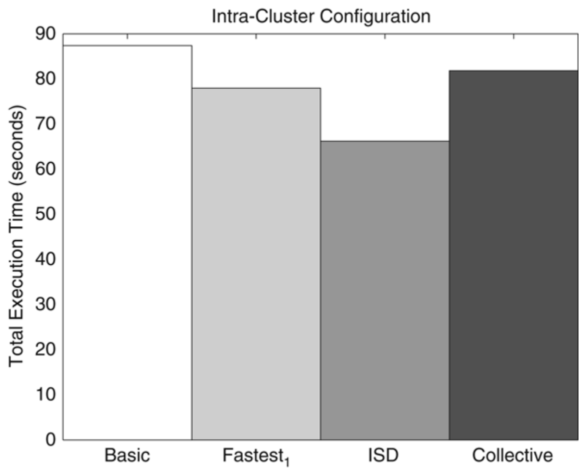

Fig. 2. Intra-cluster results.

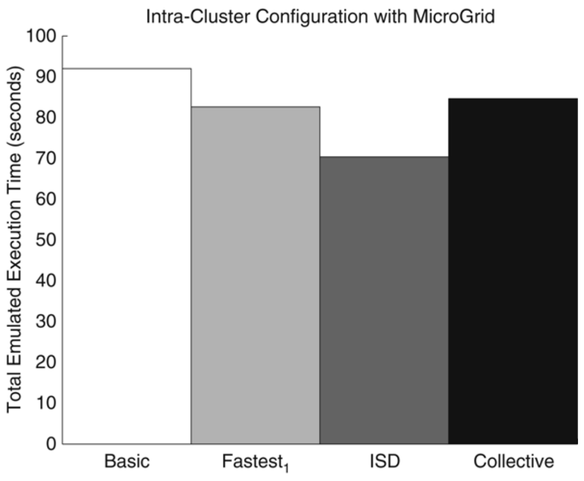

Fig. 3. Microgrid emulated intra-cluster results.

and form parallel data at the clients with block-cyclic distribution of block size 100. The bandwidth obtained for downloading a GCD block ($14\,000 \times 50$, i.e. 5.6 MB) of this matrix on a dedicated link of the Intel cluster was 38 Mbps.

The results are shown in Fig. 2. As shown in the Figure, in a tightly-coupled system like a single cluster environment, algorithms that consider loads on the server due to simultaneous downloads from the clients, namely, $fastest_1$ and ISD algorithms, give good performance. When the replica servers and the clients are located in the same set of machines, there are high interferences between simultaneous downloads and hence the two algorithms that consider these interferences for server selection perform the best. Among these two algorithms, the *ISD* algorithm due to its sophisticated techniques of weight-based impact calculations perform at least 17.72% better than the $fastest_1$.

The links between the servers and the clients are the same as the links between the clients in a tightly-coupled cluster environment. Collective download algorithm, which is tuned for scenarios when there is significant performance difference between the server–client networks and client–client networks, does not give good performance for the intra-cluster configuration. Due to the two-phase data transmissions in collective downloads, its performance is even worse than $fastest_1$. Similar to multiple server–single client scenario, the basic downloading algorithm continues to give poor performance for the multiple server–multiple client scenario. The high congestion in the network due to simultaneous data transfers using multiple threads to a single client along with the cost of maintaining shared queues lead to performance difference of at least 25% between the basic downloading algorithm and the ISD approach.

3.2. Microgrid validation

For this experiment, Microgrid was executed on 4 nodes of the 8-node Intel cluster and was used to emulate the complete 8-node cluster. The same experiment corresponding to intra-cluster configuration shown in Section 3.1 was repeated on the Microgrid emulated framework. The emulated results are shown in Fig. 3. Comparing the results in Fig. 3 with the results in Fig. 2, we find that the use of Microgrid emulation does not alter the relative performance of the different algorithms. Hence, we use Microgrid emulation for our subsequent experiments.

3.3. Inter-cluster configuration

In this configuration, 5 parallel replicas were used, each belonging to a site. Each replica corresponded to parallel data for a matrix of size $14\,000 \times 14\,000$ and distributed across 4 machines in a site. The replicas were located in UCSD, UIUC, UK, Thailand and India and the block sizes used for block-cyclic distribution in the replicas were 50, 150, 200, 250 and 300, respectively. The parallel client was run with different downloading algorithms on 4 machines of UT to download data from the replica sites and form parallel data at the clients with block-cyclic distribution of block size 100. The results are shown in Fig. 4.

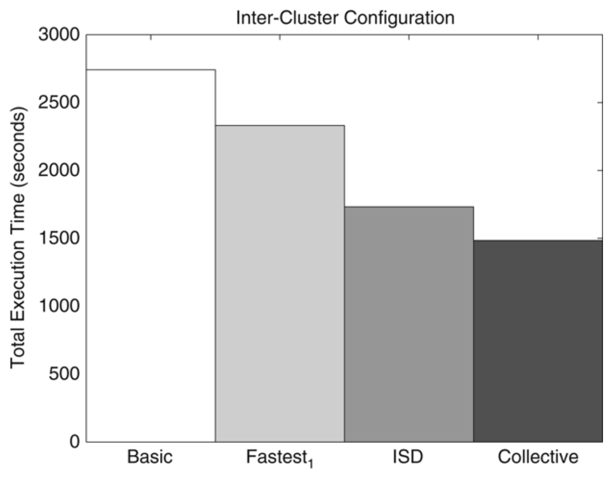

Fig. 4. Inter-cluster results.

Table 2
Number of blocks downloaded from each site for different algorithms

Algorithms/sites	UCSD	UIUC	UK	Th	India
Basic	128	21	33	75	23
Fastest$_1$	142	17	38	79	4
ISD	161	10	35	74	0
Collective	280	0	0	0	0

In order to understand the performance difference between the different algorithms, we observed the number of blocks downloaded from each replica sites by the clients in UT for each of the algorithms. Table 2 shows the site-wise distribution of the number of blocks downloaded from the site to the client machines.

As can be seen from inter-site bandwidths and latencies in the columns corresponding to UT in Table 1, the communication speeds of India–UT, and UIUC–UT links are low, the speeds of Thailand–UT and UK–UT links is moderate while the speed of UCSD–UT link is the best. Hence downloading strategies should download more data from the good replica sites (UCSD, Thailand and UK) and less data from the poor replica sites (India and UIUC). The basic downloading algorithm gives the worst performance among all the algorithms as shown in Fig. 4. In this algorithm, separate threads are maintained for each replica server and hence data will be downloaded from all the replica servers irrespective of the distance between the replica servers and the clients. Thus it downloads more data from the poor servers than the other algorithms.

The fastest$_1$ algorithm is able to reduce the number of downloads from the bad replica sites since it considers bandwidths and latencies on the links between the sites in order to select the server for downloading a block. But the fastest$_1$ algorithm gives poor performance (37% worse than collective download) in inter-cluster configuration since it makes assumptions about the impact of simultaneous downloads, especially the assumption of round-robin scheduling of network traffic at the server. In inter-cluster configuration, the scheduling of network traffic on the WAN links follows multiple complicated dynamics.

Though the ISD algorithm gives 15% less downloading performance than the collective download, it is competitive for inter-cluster configuration since its calculation of impact of simultaneous downloads on a single download is generic and valid for the WAN-based data transfers in the inter-cluster configuration. Compared to fastest$_1$, ISD downloads significantly more number of blocks from the best server (UCSD). Due to its dynamic impact calculation strategy, ISD is able to automatically deduct that the interference between simultaneous downloads from UCSD to UT is not as high or pessimistic as calculated by fastest$_1$. Accordingly, it is able to completely eliminate one bad server (India) for downloading and is also able to reduce the number of downloads from another bad server (UIUC). The number of downloads from moderate servers (UK and Thailand) are almost the same as fastest$_1$ since the ISD algorithm begins with the same assumptions as fastest$_1$ (equal impacts) and for moderate servers, the actual impacts on downloads are determined only at later stages of the algorithm.

The collective download algorithm, as shown in the previous section, downloads data from only one replica, and in our inter-cluster experiment, chooses the replica in UCSD since this is the best replica server. For our experiment, it turned out that downloading all blocks from the best replica is better than downloading some of the blocks from other replicas. Hence the collective download algorithm gives the best performance among all the algorithms. In inter-cluster configuration where clients and servers are located at different sites, the network characteristics of the links between the 5 sites (servers) and UT (clients) are much worse than the network characteristics of the links between the clients. Hence, collective download, tuned for handling this particular case, gives the best performance among all the algorithms as shown in the Figure.

3.4. *Chaotic configuration*

In this configuration, 24 machines located in 6 sites, used in the previous configuration, was used. 21 different replicas, each for matrix size of $14\,000 \times 14\,000$, were generated.[2] Each of the 21 replicas corresponded to a set of replica servers where the matrix was distributed with block-cyclic distribution of a particular block size. The number of servers in the set for a replica were randomly generated between 2 and 24 and the actual servers in the set were randomly chosen out of the 24 machines. The block size for a replica was randomly chosen from 50, 100, 150, 200, 250 and 300.

Initially, a replica was randomly generated on a set of replica servers with a particular block size. Then, a set of machines was randomly chosen with another block size for block-cyclic distribution. The parallel client program with the four data download algorithms was run on these client machines to download the data from the first set of replica servers to

[2] We were not able to generate more than 21 replicas due to insufficient disk space on our 8-node cluster.

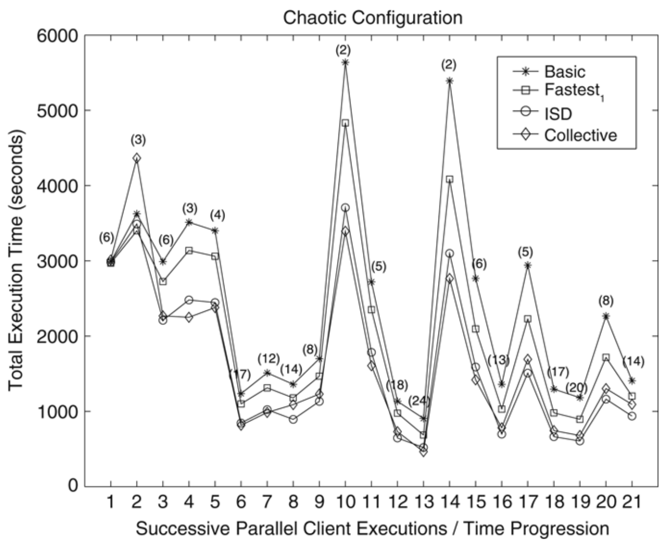

Fig. 5. Performance of algorithms with increasing number of data replicas.

the client machines. The times taken for the four algorithms were noted. The client machines were then added as the second replica set to the replica list. A new set of machines and a new block size was used for running another parallel client experiment with the four algorithms to download data from the first 2 replica servers. This process was repeated 21 times. Thus, for running the ith client application with the four different download algorithms, i data replicas were made available. This situation is similar to the usage in a grid environment where different users execute their parallel applications at different points of times leading to multiple parallel data/replicas and the number of such replicas increase over a period of time.

Fig. 5 shows the performance of the 4 different downloading algorithms for each of the 21 parallel client applications. The number shown in parentheses in the graph represent the number of machines used in the parallel client.

The figure shows a number of peaks. These correspond to small number of machines used in the corresponding client application. Smaller number of machines lead to lesser parallelism in data downloads and hence larger execution times taken for the algorithms. For data set 2, we find that the collective download took longer time than even the basic download algorithm. 2 replicas were available for the second client parallel application. The first replica was distributed on 4 machines in UIUC, UK, UCSD and India while the second replica was distributed on 6 machines in UK, UIUC, UCSD and UT. Thus both the replicas were distributed on 4 sites each. As explained in the previous section, the collective download algorithm uses only one of the available replicas for download while the other algorithms can potentially download data from many replicas. For the second client application, collective download algorithms used the second replica. Thus, collective download was able to make use of only 4 sites while the other download algorithms made use of the union of the sites in the first 2 replicas, namely, 5 sites.

We also find that collective download gave significantly best results when small number of machines were involved in the client applications (data sets 4, 10, 11, 14, 15). When small number of client machines are involved, there is small amount of simultaneous downloads from a single server to multiple clients. The ISD algorithm that is particularly suitable when the impact of simultaneous downloads on a download is high, has very few opportunities to choose the right server based on impacts of simultaneous downloads when the number of client machines are small. Thus, in these data sets, the collective algorithm that has a simple strategy for data downloads involving multiple sites, gave better performance than ISD.

When small number of data replicas were available (<16), the ISD algorithm gave significant best results when the number of client machines located in a single site/cluster were high (data sets 8 and 12). For example, out of the 14 machines used in the eighth client application, 4 machines were located in UCSD, and 4 machines in UIUC. When large number of machines are located in a cluster for a client application, significant number of shared links are involved during data transfers between the remote servers and the client machines in the cluster. Hence the amount of interference due to simultaneous data transfers between the remote servers and the client machines in the cluster are high. Hence the ISD algorithm that considers the impact of simultaneous download gave better performance than collective download in these cases.

When the number of available data replicas are high (>16), we find that the ISD algorithm consistently gave best performance. As the number of replicas are increased, the chances of finding and downloading from nearby replicas increase. When the number of replicas are sufficiently large (in our experiments, it is 16), the nearby data can be found within the same site/cluster as the client machines. This leads to large number of shared resources during data transfers, large amount of impact due to simultaneous downloads, and hence increased performance of ISD algorithms over the other algorithms.

In order to quantitatively understand the difference between ISD and collective download algorithms for chaotic setting, we define a metric, *min_avg_bw*(*client_machines, server_list*), between the client machines and a list of machines, *server_list*. We calculate this metric by finding the averages of bandwidths between a client, *c*, in client_machines, and each machine in the server_list. We then find the minimum of these averages across all the clients to form min_avg_bw(client_machines, server_list) since the total download time for data transfers between machines in server_list and client_machines is impacted by the client with the slowest links to the other machines.

For ISD algorithm, we calculate the server_list, for a given client configuration, as the union of sets of replica servers corresponding to all the previous replicas. This is because ISD algorithm downloads data from multiple replica server sets. We then denote the min_avg_bw for the server_list as *isd_bw*. For collective download algorithm, for each existing replica, we consider the replica server set corresponding to the replica as the server_list. We then find the min_avg_bw(client_machines, server_list) between the client machines and the server_list. We then find the maximum of the min_avg_bw across all replicas. This is because, unlike the ISD algorithm, the collective download algorithm downloads data only from one replica server set that is closest to the client machines. We denote the

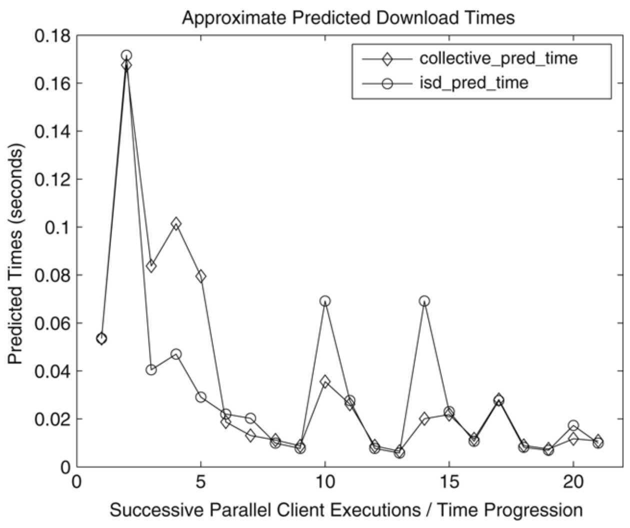

Fig. 6. Comparing ISD and collective algorithms in terms of approximate predicted times.

maximum value as the *coll_bw*. We then calculate the following approximate predicted times for ISD and collective downloads:

$$isd\_pred\_time = 1/(client\_mc\_count \times isd\_bw) \tag{5}$$

$$collective\_pred\_time = 1/(client\_mc\_count \times coll\_bw). \tag{6}$$

The client_mc_count in the equations denotes the number of client machines and takes into account the amount of parallelism, i.e. more the number of client machines, more the parallelism and less the download times as seen in Fig. 5.

Fig. 6 plots the isd_pred_time and the collective_pred_time for the same 21 data points shown in Fig. 5. The bandwidths in Table 1 were used to calculate the approximate predicted times.

Comparing Fig. 6 with Fig. 5, we find that except for few points, the relative performance between the 2 algorithms in terms of the approximate predicted times as calculated by Eqs. (5) and (6) closely correlates with the relative performance between the 2 algorithms in terms of the actual execution times. This confirms our hypothesis that the collective algorithm performs the best when a single replica server set is closer to the client machines than the other replica sets while the ISD algorithm performs the best when all the existing replica server sets are equally close to the client machines and when the aggregate bandwidth due to downloading from multiple replica sets is higher. Thus, based on the Eqs. (5) and (6), we can predict and use the best algorithm under a given set of conditions.

4. Related work

The Globus Toolkit, GT4 [11] has interfaces for replica cataloging, selection and management [6]. Various replica management architectures have been developed for managing replica placement [18,21] and efficient scheduling of computation and data in data grids [22,26]. The work by Vazhkudai et al. [30] describes a generic prototype and various steps needed for replica selection.

The efforts by Plank et al. [20] and Collins and Plank [10] deal with replicas of data segments that are scattered on wide-area resources and selection of appropriate data sources to improve the performance of data transfers. Various algorithms for replica selection were developed and evaluated. The main purpose of their algorithms is to select appropriate data sources and download from multiple data servers to a single client resource. Similar to these efforts, the work by Yang et al. [33] proposes a recursively adjusting coallocation scheme for parallel downloads from multiple replica servers to a single client. This problem of data transfers from multiple servers to a single client is referred to as Plank–Beck problem [3]. This is useful in cases like downloading music file segments and playing continuous music on a single client resource. Our algorithms are mainly aimed for transferring parallel data segments from multiple servers to multiple clients for execution of parallel numerical application on the clients. The challenges in multiple server–multiple client scenario are greater since the server selection and data download on a single client can impact the server selection and performance of data transfer on another client.

The work by Collins and Plank [10] deals with four parameters that can potentially impact the performance of download algorithms: the number of simultaneous downloads, the degree of work replication, the failover strategy and the server selection algorithm. Of these, the number of simultaneous downloads and the server selection algorithm were found to be the two most important parameters. Our work primarily deals with server selection algorithms. Our four algorithms place no restrictions on the number of simultaneous downloads. This can lead to more number of simultaneous downloads than the optimal number. However, the results in their work [10] show that there is negligible performance difference ($<2\%$) between maximum number of simultaneous downloads and the optimal number. The other 2 parameters, namely, the degree of work replication and failover strategy are used to ensure reliability, guaranteed response times and fault-tolerance. Our algorithms will be extended to encapsulate these parameters in the future.

The work by Santos-Neto et al. [23] considers coscheduling of computational tasks and data transfers. For data source selection, they use a metric called storage affinity of a task to a site, which is the amount of data needed by the task that is contained in the site. They choose the task with the largest storage affinity and schedules it on a computation resource with the data source that can provide the largest storage affinity value. Thus their strategy can lead to scenarios where a single data host can be used by multiple tasks, increasing contention to the data host. Their strategy is static in allocating the data host for a task. The work by Venugopal and Buyya [31] chooses a set of data replica hosts and computational resources for a set of sequential data-intensive jobs. For each job, they employ a set coverage-based mapping heuristic for choosing the least number of data hosts that contain the data sets needed by the job. They then find the minimum completion time (MCT) of the job by evaluating different computational resources for the chosen data hosts in terms of completion times. They then employ the MinMin heuristic for choosing the next job among the set of jobs in order to minimize the total makespan for all the jobs. However, in their work, the choice of a data host containing a data set needed by a job is static. In our work, the data sets for a job/client is dynamically chosen depending on the contention caused by the downloads from other clients.

The work by Khanna et al. considered scheduling a batch of data-intensive jobs with batch-shared I/O behavior [14,15] and also online scheduling of file-shared data-intensive jobs [12]. They employed hypergraph partitioning to schedule jobs for reducing remote I/O operations for file transfers and assuming unlimited disk cache sizes for compute nodes [12,15]. They later developed a scheduling strategy based on 0–1 Integer Programming and another scheduling strategy using bi-level hypergraph partitioning to also consider replica placements and selections with restrictions on disk cache sizes for compute nodes [14]. In a more recent work [13], they have considered the problem of scheduling a set of file transfers from one of multiple possible sources/replicas to a set of destination nodes across heterogeneous clusters to minimize the overall file transfer completion time. Their methods also try to minimize contention and maximize the parallelism in data transfers. They propose two scheduling algorithms, one based on 0–1 IP and the other based on max-weighted graph matching. In all these efforts, they deal with accesses and transfers of entire files needed for their independent tasks. In our work, each task of a parallel application deals with segments of files. They also assume single port model for data transfers where multiple download requests to the same node are serialized and also uniform contention on all links to a storage node. Our work is generic and dynamically determines the weights for contentions on different links for data transfers based on observations. Moreover, their scheduling methodologies for heterogeneous clusters [13] need network topology as one of the inputs. In a dynamic grid environment, obtaining and updating network topologies are non-trivial. Our work does not assume knowledge of network topologies.

5. Conclusions

In this work, we have developed different algorithms for the efficient selection of parallel data servers out of different data replicas and efficient transfer of the parallel data from multiple servers to multiple clients. The challenges involved are much more complicated than for selection and transfer of data from multiple servers to a single client for which algorithms already exist. Apart from extending two basic algorithms to our problem, we have also developed two new advanced algorithms, namely, algorithm based on Impact of Simultaneous Downloads (ISD), and collective download. The relative performance of the algorithms depend on the network characteristics of the links between the servers and the clients and between the clients themselves. While collective download algorithm gives good performance when the difference in network characteristics is large and for small number of client machines, the ISD algorithms give best performance when large number of client machines are located in a single cluster/site and when the number of replicas are large. In all cases, the collective and ISD algorithms performed 15%–30% better than the two basic algorithms.

6. Future work

The techniques mentioned in this paper will be integrated into a real grid system, GrADSolve [28] which follows application-level scheduling for selecting computational resources. A 3-way comparison will then be made between: 1. when computational resources are chosen by the GrADSolve scheduler and sequential data from the user is segmented and distributed to computational resources, 2. when computational resources are chosen by the scheduler, parallel data is already available in data replica servers and data is moved from multiple servers to multiple client resources, and 3. when computation is performed on the "best" servers that already contain the parallel data, i.e. when computation is moved to data.

Acknowledgments

The authors would like to thank researchers Thara Angskun (Kasetsart University, Thailand), Balajee Kannan (University of Tennessee, USA) and Sachin Wasnik (University of Belfast, UK) for their inputs on bandwidth and latency numbers between the different sites. Thanks are also due to the anonymous reviewers for their valuable inputs that helped in improving the quality of the paper.

References

[1] B. Allcock, J. Bester, J. Bresnahan, A.L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, S. Tuecke, Data management and transfer in high performance computational grid environments, Parallel Computing Journal 28 (5) (2002) 749–771.

[2] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, B. Toonen, Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus, in: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing, 2001.

[3] M. Allen, R. Wolski, The livny and Plank–Beck problems: Studies in data movement on the computational grid, in: Supercomputing 2003, Phoenix, Arizona, USA, 2003.

[4] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, Y. Robert, Centralized versus distributed schedulers for multiple bag-of-task applications, in: 20th International Parallel and Distributed Processing Symposium, 2006.

[5] F. Berman, R. Wolski, The apples project: A status report, in: Proceedings of the 8th NEC Research Symposium.

[6] M. Cai, A. Chervenak, M. Frank, A peer-to-peer replica location service based on a distributed hash table, in: Proceedings of the SC2004 Conference, 2004.

[7] R.-S. Chang, P.-H. Chen, Complete and fragmented replica selection and retrieval in data grids, Future Generation Computer Systems 23 (4) (2007) 536–546.

[8] A. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, R. Schwartzkopf, Performance and scalability of a replica location service, in: Proceedings of 13th IEEE International Symposium on High Performance Distributed Computing, 2004.

[9] The compact muon solenoid experiment. http://cmsinfo.cern.ch.

[10] R.L. Collins, J.S. Plank, Downloading replicated, wide-area files — a framework and empirical evaluation, in: 3rd IEEE International Symposium on Network Computing and Applications, NCA-2004, Cambridge, MA, 2004.

[11] The globus project. http://www.globus.org.
[12] G. Khanna, U. Catalyurek, T. Kurc, P. Sadayappan, J. Saltz, A data locality aware online scheduling approach for i/o-intensive jobs with file sharing, in: Proceedings of the 12th Workshop on Job Scheduling Strategies for Parallel Processing, JSSPP, 2006.
[13] G. Khanna, U. Catalyurek, T. Kurc, P. Sadayappan, J. Saltz, Scheduling file transfers for data-intensive jobs on heterogeneous clusters, in: Proceedings of the 13th European Conference on Parallel and Distributed Computing, Europar, 2007.
[14] G. Khanna, N. Vydyanathan, U. Catalyurek, T. Kurc, S. Krishnamoorthy, P. Sadayappan, J. Saltz, Task scheduling and file replication for data-intensive jobs with batch-shared I/O, in: HPDC '06: Proceedings of the 15th International Symposium on High performance Distributed Computing, 2006.
[15] G. Khanna, N. Vydyanathan, T. Kurc, U. Catalyurek, P. Wyckoff, J. Saltz, P. Sadayappan, A hypergraph partitioning based approach for scheduling of tasks with batch-shared I/O, in: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid, CCGrid, 2005.
[16] X. Liu, A. Chien, Realistic large-scale online network simulation, in: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, Conference on High Performance Networking and Computing, Pittsburgh, Pennsylvania, USA, 2004.
[17] X. Liu, H. Xia, A. Chien, Validating and scaling the microgrid: A scientific instrument for grid dynamics, Journal of Grid Computing II (2) (2004) 141–161.
[18] Y. Machida, S. Takizawa, H. Nakada, S. Matsuoka, Intelligent data staging with overlapped execution of grid applications, Future Generation Computer Systems. doi:10.1016/j.future.2007.07.005.
[19] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, S. Vadhiyar, Numerical libraries and the grid: The GrADS experiments with ScaLAPACK, Journal of High Performance Applications and Supercomputing 15 (4) (2001) 359–374.
[20] J.S. Plank, S. Atchley, Y. Ding, M. Beck, Algorithms for high performance, wide-area distributed file downloads, Parallel Processing Letters 13 (2) (2003) 207–224.
[21] K. Ranganathan, I. Foster, Identifying dynamic replication strategies for a high-performance data grid, in: GRID '01: Proceedings of the Second International Workshop on Grid Computing, 2001.
[22] K. Ranganathan, I. Foster, Decoupling computation and data scheduling in distributed data-intensive applications, in: HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, 2002.
[23] E. Santos-Neto, W. Cirne, F. Brasileiro, A. Lima, Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids, in: Proceedings of the 10th International Workshop on Job Scheduling Strategies for Parallel Processing, JSSPP 2004, 2004.
[24] Scalable linear algebra package (ScaLAPACK). http://www.netlib.org/scalapack.
[25] H.J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, A. Chien, The microgrid: A scientific tool for modeling computational grids, in: Proceedings of the IEEE/ACM SC2000 Conference, Dallas, TX, USA, 2000.
[26] M. Tang, B.-S. Lee, X. Tang, C.-K. Yeo, The impact of data replication on job scheduling performance in the data grid, Future Generation Computer Systems 22 (3) (2006) 254–268.
[27] R. Thakur, W. Gropp, E. Lusk, A case for using mpi's derived datatypes to improve I/O performance, in: Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing, CDROM, 1998.
[28] S. Vadhiyar, J. Dongarra, GrADSolve — A grid-based rpc system for parallel computing with application-level scheduling, Journal of Parallel and Distributed Computing 64 (2004) 774–783.
[29] S. Vazhkudai, J. Schopf, I. Foster, Predicting the performance of wide area data transfers, in: Proceedings of the 16th international Parallel and Distributed Processing Symposium, 2002.
[30] S. Vazhkudai, S. Tuecke, I. Foster, Replica selection in the globus data grid, in: Proceedings of the First IEEE/ACM International Conference on Cluster Computing and the Grid, CCGRID 2001, 2001.
[31] S. Venugopal, R. Buyya, A set coverage-based mapping heuristic for scheduling distributed data-intensive applications on global grids, in: 7th IEEE/ACM International Conference on Grid Computing, Grid 2006, 2006.
[32] R. Wolski, N. Spring, J. Hayes, The network weather service: A distributed resource performance forecasting service for metacomputing, Journal of Future Generation Computing Systems 15 (5–6) (1999) 757–768.
[33] C.-T. Yang, I.-H. Yang, S.-Y. Wang, C.-H. Hsu, K.-C. Li, A recursively-adjusting co-allocation scheme with a cyber-transformer in data grids, Future Generation Computer Systems. doi:10.1016/j.future.2006.11.005.

Sandip Tikar obtained his Bachelors degree from the department of Computer Science, Pune University, India in 2002, and received his Masters degree from Supercomputer Education and Research Centre, Indian Institute of Science, India in 2006. He is currently associated with Evergrid, Pune as a Software Engineer. His areas of interest include replica management, computational economy, checkpointing and migration of parallel applications over Grids.

Sathish Vadhiyar is an Assistant Professor at Supercomputer Education and Research Centre (SERC), Indian Institute of Science. He obtained his Bachelors degree from the Department of Computer Science and Engineering, Thiagarajar College of Engineering, India in 1997. He received his Masters degree in Computer Science from Clemson University, USA in 1999. He graduated with a Ph.D. from the Computer Science Department, University of Tennessee, USA in 2003. His doctoral research involved building scheduling mechanisms for Grid applications. Dr. Vadhiyar's research interests include MPI collective communications, scheduling and rescheduling methodologies for Grid systems, performance modeling of parallel applications, and Grid applications in the areas of DNA sequence evolutions and climate modeling. He has published papers in the area of parallel, distributed and Grid computing.