An Efficient MPI_Allgather for Grids [*]

Rakhi Gupta
Computer Science/Information Technology Department
Jaypee Institute of Information Technology University
Noida-201307
India
rakhi.hemani@jiit.ac.in

Sathish Vadhiyar
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore-560012
India
vss@serc.iisc.ernet.in

ABSTRACT

Allgather is an important MPI collective communication. Most of the algorithms for allgather have been designed for homogeneous and tightly coupled systems. The existing algorithms for allgather on Grid systems do not efficiently utilize the bandwidths available on slow wide-area links of the grid. In this paper, we present an algorithm for allgather on grids that efficiently utilizes wide-area bandwidths and is also wide-area optimal. Our algorithm is also adaptive to grid load dynamics since it considers transient network characteristics for dividing the nodes into clusters. Our experiments on a real-grid setup consisting of 3 sites show that our algorithm gives an average performance improvement of 52% over existing strategies.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel programming*

General Terms

Algorithms, Design, Performance

Keywords

MPI_Allgather, Grids

1. INTRODUCTION

Collective communication operations are widely used in MPI applications and play an important role in their performance. Hence, various projects have focused on optimization of collective communications for various kinds of parallel computing environments including homogeneous and heterogeneous LAN networks [3,4,6,8,22] and most recently Grid systems [9,11,12,14,19,20].

Allgather is an important many-to-many collective communication operation. Allgather on N processes can be considered as equivalent to N broadcasts of data, each conducted with a distinct root. An algorithm for implementing allgather needs to specify a schedule of the required data transfers. An efficient allgather schedule is dependent on various factors including message size and network characteristics.

Most of the algorithms for allgather are designed for homogeneous networks [3, 4, 6, 8, 22]. These algorithms follow uniform communication patterns between all nodes and hence cannot be used in Grid settings where the network links are highly heterogeneous and the link characteristics change over time. Current popular techniques for allgather on grids [12,14,18] follow static network hierarchical schemes, where the nodes are divided into clusters on the basis of network topology and a representative/coordinator node is chosen from each cluster. The inter cluster communications are done through these representative nodes. At any given point of time, only one data transfer takes place between two representative nodes. These techniques are not efficient since the clusters are separated by wide-area links that can sustain multiple simultaneous data transfers with the same end-to-end bandwidths [5]. Hence the existing strategies for grids do not exploit the total available bandwidths of the wide-area links.

In this paper, we present our cluster based and incremental greedy algorithm called Min^3-Allgather, for allgather on grids. Our algorithm divides the nodes into clusters based on transient network characteristics, namely available bandwidths, and follows a recursive approach where allgather is performed at different levels of the hierarchy. Our algorithm allows multiple simultaneous communications between 2 clusters separated by slow wide-area links and hence effectively utilizes the available bandwidths of the wide-area links. Our algorithm is also wide-area optimal [12,14] since it ensures that a data segment is transferred only once between two clusters separated by a wide-area link. We compared the time taken by allgather schedules determined by this algorithm with current popular implementations. We also compared our algorithm with a strategy where allgather is constructed from a set of broadcast trees. Our experiments on a real-grid setup show that the average performance improvement of our algorithm is 52% over existing strategies.

In Section 2, we present related efforts in the development

[*]This work is supported by Indian Institute of Science's 10th Plan Grant SERC Part(2A) Special Grant (45/SERC)

of allgather algorithms. In Section 3, we describe the design principles used in the development of our algorithm. Section 4 explains the communication models used in our algorithm. Section 5 describes our algorithm, Min^3-Allgather, for grids. In Section 6, we compare our algorithm with existing strategies on a real-grid setup and present results. Section 7 gives conclusions and Section 8 presents future work.

2. RELATED WORK

A number of generic and theoretically efficient allgather algorithms have been developed for homogeneous clusters [8]. *Simple* algorithm posts all the sends and receives and waits for their completion. In this algorithm, a process sends messages to other processes in the order of their ranks. In order to avoid the potential node and network contention caused by the simple algorithm, *spreading simple* algorithm was proposed. In this algorithm, in each iteration i, a process p sends its data to process (p+i) mod N and receives data from process (p-i+N) mod N where N is the number of processes. The *ring/bucket/circular* algorithm [6] was developed for architectures where near-neighbor communications can be beneficial. At each iteration i, a process sends data corresponding to index (p-i+1+N) mod N to its right neighbor process, i.e. process (p+1) mod N, and receives data from its left neighbor process, i.e. process (p-1) mod N. The time taken for allgather in simple, spreading simple and ring algorithms are (N-1)*(L + m/B), where L is the latency, m is the message size and B is the bandwidth.

Recursive doubling algorithm takes lesser time as compared to previous algorithms because the number of transfers (hence latency) is reduced. The number of iterations required when N is a power of 2 is log N. In each iteration i, processes separated by a distance of 2^{i-1} exchange data. Recursive doubling algorithm is sub-optimal when the number of processes is not a power of two, because some data transfers may be repeated. The *dissemination* algorithm developed by Benson et. al. [3] is similar to the single port algorithm described in work by Bruck et. al. [4]. The algorithm has $\lceil(\log N)\rceil$ iterations. In each iteration i, process p sends data to the process (p + 2^{i-1}) mod N. The amount of data sent in all iterations, (except for the last) is 2^{i-1}*m. For the last iteration i, (p - 2^{i-1})*m data is sent. The work by Thakur et. al. [22] gives a detailed analysis of the above algorithms for allgather. It is shown that different algorithms are optimal for different message sizes. MPICH [17], the popular implementation of MPI, uses recursive doubling for small message sizes when number of processes is a power of 2. However, if the number of processes is not a power of 2, dissemination algorithm is used. MPICH uses ring algorithm for large message sizes.

Current popular topology-aware allgather scheduling strategies for grids divide the network into network hierarchies. The nodes are divided into clusters and a coordinator node is assigned to each cluster. MagPIe [12] proposes a three phase algorithm for allgather - gather data at coordinators, allgather among coordinators and broadcast of data by coordinators. In the second phase, the coordinators perform allgather using spreading simple algorithm. At this stage all coordinators have all the required data. In the third phase, coordinators broadcast data using binomial broadcast to processes in the cluster. MPICH-G2 [18] implements a similar algorithm. The algorithm has 2 phases. In the first phase, data is gathered up the hierarchy using recursive doubling. Next, data is broadcast downwards using binomial broadcast. The major drawback of these approaches is that data transmission is sequentialized at coordinators. This results in low usage of available bandwidths at higher layers of hierarchy (e.g. WAN links). Also, in these approaches, the network hierarchy for a given grid setup is formed on the basis of information about WAN and LAN links. Hence the network hierarchy is static and the same hierarchy will be used for all the allgather operations. Our algorithm allows multiple simultaneous data transfers between 2 clusters resulting in increased use of available bandwidths on wide-area networks and hence improved performance of allgather on grids. Also, our algorithm forms the network hierarchy on the basis of transient network characteristics, namely, available bandwidths. Thus our algorithm is adaptive to grid load dynamics since the network hierarchy can change with the changes in transient network characteristics.

3. DESIGN PRINCIPLES

Following are the design principles used in the construction of our allgather algorithm.

1. **Multi-level collective communication algorithms are more suitable for grids than single-level algorithms.**

   Collective communication algorithms for homogeneous systems [8,17] and some recent heuristics for distributed systems [16] follow a single-level strategy for build communication schedules. The recent efforts for grids [9, 11,12,14] divide the given set of nodes into clusters/pools, form hierarchies between the clusters and follow different strategies for different levels of hierarchies. The intra-cluster links (LAN links, high performance networks) are relatively faster than inter-cluster (WAN, internet, campus networks) links. Such clustering of nodes helps in designing algorithms that carefully avoid transmitting the same data multiple times on a slow link connecting two clusters. This is essential for ensuring wide area optimality of the collective communication algorithms [14]. While some strategies divide the nodes or network based on static network topologies [11,12,14], we divide the nodes based on transient network characteristics similar to our earlier approach for broadcasts [9].

2. **WAN links can sustain many simultaneous transfers without performance degradation**

   In the current popular algorithms for allgather on grids links [12,18], a single coordinator node is chosen in every cluster and only the coordinators participate in inter-cluster data transfers across WAN links. However, it is difficult to utilize total bandwidth available on a WAN link by a single data transfer. This is because of TCP behavior, where a host sends some packets and then waits for acknowledgment before sending next packets. High Round Trip Times (RTT) between nodes separated by WAN links cause late arrival of acknowledgment packets resulting in delays in the transmission of packets, and hence lesser bandwidth utilization. It has been found that many simultaneous transfers on these links can help in effective utilization of underlying bandwidth [5].

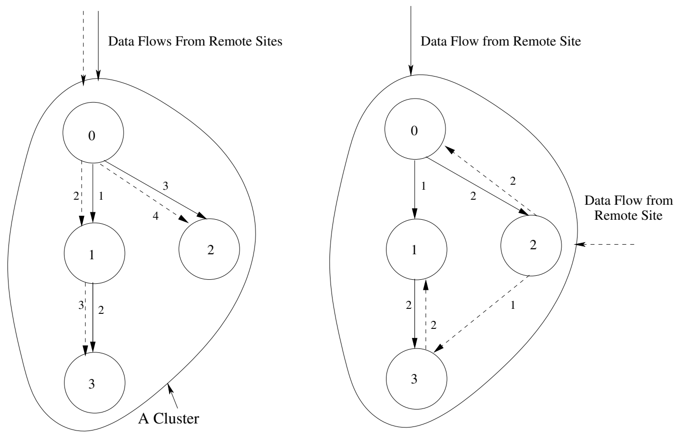

Data Flows From Remote Sites
0
1
2
3
A Cluster
Data Flow from Remote Site
Data Flow from Remote Site

Figure 1: Broadcast Trees within a Cluster

Moreover, the concept of choosing coordinator nodes for data transfers in WAN links is beneficial for broadcast communications where only a single message has to propagate to all nodes. In this case, choosing multiple nodes in a cluster to send messages to nodes in another cluster will lead to poor performance of broadcast operations. However, in allgather operation, different nodes in one cluster have distinct messages to send to nodes in another cluster. Making the nodes send these distinct messages to the coordinator nodes will lead to severe sequentialization of messages and act as bottlenecks in communications. In our algorithm, we allow multiple simultaneous inter-cluster transfers between any two clusters separated by a WAN link.

3. **Scheduling of current data transfers should be based on previously scheduled data transfers.**

   In an allgather operation, a single node may be involved in multiple data transfers corresponding to different messages from different sources. The data transfers in allgather must be scheduled such that there is enough parallelization in the work performed by different nodes in different time steps. This helps in avoiding node-level bottlenecks that can be caused when processing multiple messages received by a node in a single time step. Care must also be taken such that different nodes perform equivalent amount of work in different time steps. For example, Figure 1 depicts a situation where a cluster has to broadcast two distinct messages obtained from remote sites during an allgather operation. Two possible methods of broadcasts are shown. The first method of broadcasts uses the same tree for both broadcasts, whereas the second uses two different trees. Assuming a homogeneous, fully connected network, unit transfer time for data, single port full duplex connections (i.e. a host can do at-most 1 send and receive simultaneously) and a condition whereby a host can send data only if the previous send is complete, the first method completes the broadcasts in '4' time units and the second method completes broadcasts in '2' time units. Note that the numbers along the data transfers (arrows) indicate the time for completion of data transfer. It is easy to observe that the second method carefully avoids sequentialization of communication operations within the nodes. Thus the communication schedules for allgather have to be incrementally built by taking into account the already scheduled data transfers to avoid node-level bottlenecks.

4. COMMUNICATION MODELING

To estimate the time taken for an allgather operation and for scheduling the next data transfer, the individual data transfers need to be modeled. For modeling data transfers, we use 4 network parameters corresponding to a message size *m* - *latency* (L), *bandwidth* (b), *overhead Send* (os(m)) and *gap* (g(m)). The gap parameter is defined as the minimum time interval between consecutive message transmissions or receptions [7]. We use parametrized-LogP benchmark [13] to measure os(m) and g(m) and our own communication benchmark program to measure L and b.

The next aspect of modeling relates to host-specific parameters. In an allgather operation, a single host may be processing multiple messages from different source nodes. Hence it is essential to determine the time at which a host will be available for the next send and/or receive (recv). There are two popular models to determine the host available times: *single-port half-duplex* [10] and *single-port full-duplex* [2]. In the single-port half duplex model, a host can either perform 1 send or 1 recv in a single time step. In the single-port full-duplex model, the host can simultaneously perform a send and a receive in a time step. Equations 1 - 3 show the calculations for times when a host, s, will be available for the next send and receive, after sending a message,m, to host,r. Equation 2 show the calculations assuming a half-duplex model and Equation 3 show the calculations assuming a full-duplex model. The following terms are used in the equations:

- **TotalTime[m]** represents the time at which, r, receives the message.
- **TransferTime[m,s,r]** represents the time duration, for transfer of message of size m form a host, s, to a host, r.
- **StartTime[m,s,r]** represents the time corresponding to the start of transfer.
- **CommLinkAvailTime[s,r]** represents the time at which communication link from host s, to host r, is available for transfer of next message. Note that this time is independent of the time at which communication link, from host r, to host s, is available for message transfer.
- **HostAvailTime[h]** represents the time at which host, h, would be available for next send or recv. This parameter is used only in the half-duplex model.
- **HostAvailTimeForSend[s,h]** represents the time at which the host s, is ready to send a message to the host h. This parameter is used only in the full duplex model.
- **HostAvailTimeForRecv[h]** represents the time at which the host h, is ready to recv a message. This parameter is used only in the full duplex model.

$$
\begin{aligned}
TotalTime[m] &= StartTime[m,s,r]+ \\
&\quad TransferTime[m,s,r] \\
TransferTime[m,s,r] &= Latency(s,r)+ \\
&\quad \frac{messageSize(m)}{Bandwidth(s,r)}
\end{aligned} \tag{1}
$$

Single-port Half-duplex Model

$$
\begin{aligned}
StartTime[m,s,r] &= max(\\
&\quad CommLinkAvailTime[s,r], \\
&\quad HostAvailTime[s], \\
&\quad HostAvailTime[r] \\
CommLinkAvailTime[s,r] &= StartTime[m,s,r]+ \\
&\quad g(m,s,r) \\
HostAvailTime[s] &= StartTime[m,s,r]+ \\
&\quad os(m,s,r) \\
HostAvailTime[r] &= TotalTime[m]
\end{aligned} \tag{2}
$$

Single-port Full-duplex Model

$$
\begin{aligned}
StartTime[m,s,r] &= max(\\
&\quad CommLinkAvailTime[s,r], \\
&\quad HostAvailTimeForRecv[r], \\
&\quad HostAvailTimeForSend[s]) \\
CommLinkAvailTime[s,r] &= StartTime[m,s,r]+ \\
&\quad g(m,s,r) \\
HostAvailTimeForSend[s] &= StartTime[m,s,r]+ \\
&\quad os(m,s,r) \\
HostAvailTimeForRecv[r] &= TotalTime[m]
\end{aligned} \tag{3}
$$

5. MIN^3-ALLGATHER ALGORITHM

We propose a heuristic algorithm, Min^3-Allgather for generating efficient schedules for allgather on grids. The algorithm clusters the hosts participating in allgather according to the bandwidth characteristics of the links between the hosts, and identifies different levels of network hierarchies. Data transfers are then scheduled at each level of the hierarchy using a recursive, top-to-bottom approach. The following subsections give details of our algorithm.

5.1 Identification of Network Hierarchy or Formation of Pool Tree

As described in Section 3, we cluster the network and identify network hierarchies. A cluster (or pool) is a set of hosts, such that the average of bandwidths of links from a host to all other hosts is greater than or equal to some threshold bandwidth. Thus, smaller the threshold bandwidth of a pool, greater the number of hosts in the pool. The pools at high levels of network hierarchy (e.g. WANs), have a low threshold bandwidth, and can be split into pools with a higher threshold bandwidth. For example, a WAN can be split into constituent LANs. Thus a pool may be split

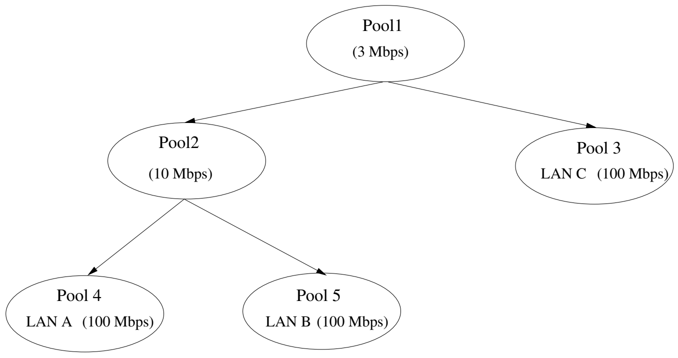


Figure 2: Example Pool Tree

recursively into sub-pools. This can be represented in the form of a *pool tree*. Figure 2 shows a pool tree for a network consisting of 3 LANs A, B and C. Pool1, also referred to as *root pool*, consists of all the hosts in the network and has the lowest threshold bandwidth - 3 Mbps. This is split into two pools, Pool2 and Pool3. Pool2 has threshold bandwidth of 10 Mbps and, Pool3 has threshold bandwidth of 100 Mbps and corresponds to LAN C. Pool2 is split into Pool4 and Pool5. These correspond to LANs A and B respectively. To calculate the bandwidth threshold values, we first obtained bandwidth values between all pairs of machines. Then, we sorted these values and created sets, such that bandwidth values in a set are in a range of 10%. For each such set, we found the maximum bandwidth value, and used the maximum values as thresholds for the formation of pool tree. This procedure of formation of pools or clusters is similar to the formation of "logical clusters" in earlier efforts by Estefanel and Mounie [1] and Lowekamp et. al. [15]. These efforts divide the network into subnets based on link latencies and throughput. Our current work is complementary to these efforts since our allgather algorithm can also use the subnets formed by these efforts.

Given such a pool tree, the algorithm starts at the root node of the pool tree, *root-pool*. Each host i belonging to the root-pool has data d_i for broadcast to other nodes. As a host can be included in only 1 sub-pool of a pool (in this case, sub-pools of the *root-pool*), only 1 sub-pool has d_i. We schedule data transfers such that every sub-pool of the input pool (*root-pool*) will contain data d_i. To ensure "wide area optimality" [14], data is transmitted exactly once to a sub-pool. This implies that only 1 host in each sub-pool of the input pool will contain d_i. However allgather operation is complete only when all hosts have the data d_i. To ensure this, we recursively schedule data transfers considering each sub-pool as an input pool. The recursion stops when sub-pools of the input pool are individual hosts.

Next we describe details of this algorithm. Section 5.2 describes the scheduling algorithm within a pool, and Section 5.3 describes the complete recursive algorithm.

5.2 Finding Allgather Schedule for Sub-Pools of an Input Pool

Given an input pool, *InputPool* and its subpools, where each host $host_i$ is contained in one of the subpools of the input pool, and has data d_i, the first step is to determine a schedule of inter-subpool data transfers such that data d_i is transferred to the other subpools. At the end of this step, one host in every subpool will have data d_i. We denote the set of d_i where i ∈ set of nodes participating in allgather as *DataSet*. We denote the host in the input pool containing data d_i as *InitSource[d_i]*. At the end of this first step, each subpool will have the *DataSet*.

We introduce the following terms:

- *Sources[d_i]* represents the set of hosts that contain d_i. Initially, this set contains only one host, InitSource[d_i].
- *DestPools[d_i]* represents the set of sub-pools of the input pool, such that no host belonging to these sub-pools, contains d_i. Initially this includes all sub-pools of the input pool excluding the sub-pool containing the source host.
- *MinTime[d_i,j]* is the minimum time by which d_i can be transmitted to a host belonging to a sub-pool j belonging to DestPools[d_i].
- *MinDest[d_i,j]* represents the host belonging to sub-pool j that can receive d_i in MinTime[d_i,j].
- *MinSrc[d_i,j]* is the id of a host in Sources[d_i], that can send d_i to MinDest[d_i,j] in MinTime[d_i,j].

Function InitSched, shown in Figure 3, shows the initial calculation of these terms. InitSource[d_i] is added to Sources[d_i] in line 3, DestPools are created in line 6. MinTime, MinSrc and MinDest corresponding to transfer of d_i to a SubPool is calculated by function FindMin.

```
1  Algorithm: InitSched()
   input  : MsgSize, InputPool, DataSet, InitSource
   output: Sources,DestPools, MinTime, MinDest,
           MinSrc
2  for $d_i \in$ DataSet do
3      add InitSource[$d_i$] to Sources[$d_i$] ;
4      for SubPool ∈ InputPool do
5          if InitSource[$d_i$] ∉ SubPool then
6              add SubPool to DestPools[$d_i$] ;
               /* let j be the index of SubPool in
                  DestPools[$d_i$]                        */
7              (MinTime[$d_i$,j], MinSrc[$d_i$,j], MinDest[$d_i$,j] )
               = FindMin ( $d_i$, Sources, SubPool, MsgSize) ;
8          end
9      end
10 end
11 return (Sources,DestPools,MinTime, MinSrc, MinDest)
   ;
```

Figure 3: InitSched()

Function FindMin, shown in Figure 4, calculates TotalTime for transfer of MsgSize data, according to equations 1-3, for each sender in Sources[d_i] and each receiver in SubPool. It then finds and returns the minimum TotalTime, MinTime, corresponding to a MinSrc in Sources[d_i] and MinDest in SubPool.

After the above initial calculations, we schedule data transfers using a greedy approach as shown in ScheduleInputPool() function in Figure 5. We identify data, *SchedData* and *SchedPool* such that MinTime[SchedData, SchedPool] is

```
1  Algorithm: FindMin()
   input  : $d_i$, Sources, SubPool, MsgSize
   output: MinTime, MinSrc, MinDest
2  MinTime = $\infty$ ;
3  for s ∈ Sources[$d_i$] do
4      for host ∈ SubPool do
           /* find TotalTime corresponding to
              transfer of MsgSize data from s to
              host                                   */
5          if TotalTime < MinTime then
6              MinTime = TotalTime ;
7              MinSrc = s ;
8              MinDest = host ;
9          end
10     end
11 end
12 return (MinTime, MinSrc, MinDest) ;
```

Figure 4: FindMin()

```
1  Algorithm:ScheduleInputPool()
   input  : MsgSize, InputPool, InitSource, DataSet,
            mode
   output: SchedFiles, Sources
2  (Sources,DestPools, MinTime, MinSrc, MinDest) =
   InitSched ( MsgSize, InputPool, InitSource, DataSet) ;
3  while TRUE do
4      SchedTime = $\infty$ ; ComeOut = TRUE ;
5      for $d_i \in$ DataSet && DestPools[$d_i$] $\neq \phi$ do
6          ComeOut = FALSE ; MinDestPoolTime = $\infty$ ;
7          for p ∈ DestPools[$d_i$] do
8              if MinTime[$d_i$,p] < MinDestPoolTime then
9                  MinDestPoolTime = MinTime[$d_i$,p] ;
                   MinDestPool = p ;
10             end
11         end
12         if MinDestPoolTime < SchedTime then
13             SchedTime = MinDestPoolTime ; SchedDest
               = MinDest[$d_i$,MinDestPool] ;
14             SchedSrc = MinSrc[i,MinDestPool] ;
               SchedPool = MinDestPool ;
15             SchedData = $d_i$ ;
16         end
17     end
18     if ComeOut == TRUE then break ;
19     ScheduleTransfer (SchedSrc, SchedDest,
       SchedData, SchedFiles ) ;
20     add SchedDest to Sources[$d_i$] ;
21     remove SchedPool from DestPools[$d_i$] ;
22     (MinTime, MinSrc, MinDest) = UpdateAfterSched
       (SchedData, SchedSrc, SchedDest , DestPools,
       Sources, MsgSize, DataSet, Mode) ;
23 end
```

Figure 5: ScheduleInputPool()

the minimum of all MinTime values returned by InitSched() function. This involves applying minimization at 3 stages[1]. In the first stage, we apply minimization in the FindMin() function to calculate the minimum time required for transfer of data d_i to a given subpool, SubPool. This is denoted by MinTime[d_i, SubPool]. In the second stage, we find the destination subpool for d_i, *MinDestPool[d_i]* such that MinTime[d_i, MinDestPool[d_i]] is minimum over all destination subpools (lines 7-11), i.e. $Min_{SubPool \in allSubPools}(MinTime[d_i, SubPool])$. In the third and final stage, we find the data, *SchedData* , such that MinTime[SchedData,MinDestPool[SchedData]] is minimum over all all values of MinTime[d_i, MinDestPool[d_i]], i.e. $Min_{d_i \in DataSet}(Min_{SubPool}(MinTime[d_i, SubPool]))$ (lines 4-17). We denote MinDestPool[SchedData] as SchedPool. We denote the source and destination hosts corresponding to MinTime[SchedData,SchedPool] as SchedSrc and SchedDest, respectively. We then schedule the data transfer of SchedData between SchedSrc and SchedDest (line 19).

After the data transfer, SchedPool is removed from DestPools[SchedData], and SchedDest is added to Sources [SchedData] (lines 20, 21). We also update the host model parameters for SchedSrc and SchedDest using equations 2 or 3, for half-duplex or full-duplex model, respectively (line 22). The function for updating the model parameters is shown in Figure 6. These updates lead to the invalidation of MinTime[d_i, pool] if any of the following conditions are true.

- d_i is equal to SchedData (lines 2-6), SchedDest is the new source of SchedData, and can be equal to MinSrc[SchedData, p] for some pool, p.
- MinSrc[d_i,pool] is equal to SchedSrc or MinDest[d_i, pool] is equal to SchedDest (line 10). As parameters for SchedSrc and SchedDest are updated, MinTime, MinSrc, MinDest for d_i and pool need to be updated.
- MinSrc[d_i,pool] is equal to SchedDest or MinDest[d_i, pool] is equal to SchedSrc and the model used is Single Port Half-Duplex (line 10). In this model, sends and receives are sequentialized at a host, hence, the times at which SchedSrc and SchedDest can receive and send messages respectively are also updated. Thus MinTime, MinSrc, MinDest for and pool need to be updated.

We check for the above conditions for all $d_i \in$ DataSet and corresponding pools $\in$ DestPools(d_i). If a condition is true, corresponding values for MinTime, MinSrc and MinDest are recomputed by calling FindMin function. Scheduling of data transfers and updates are repeated till DestPools[d_i] is ϕ for all d_i.

5.3 Finding Allgather Schedule for Pool Tree

For calculating the optimal allgather schedule for the entire pool tree, we calculate the optimal schedule for the subpools of the root pool in the pool tree using ScheduleInputPool algorithm described above. Next for each sub-pool, we identify the source host of each d_i within the sub-pool. Now we call Min^3-Allgather recursively treating each sub-pool as an input pool. The recursion stops when the sub-pools of the input pool are individual hosts. The recursive algorithm is shown in Figure 7.

[1]Hence the name Min^3-AllGather for the algorithm.

```
1  Algorithm: UpdateAfterSched()
   input  : SchedData, SchedSrc, SchedDest, DestPools,
            Sources, MsgSize, DataSet, Mode
   output: MinTime, MinSrc, MinDest
   /* Update host parameters for SchedSrc and
      SchedDest according to mode                    */
2  if DestPools[SchedData] $\neq \phi$ then
3      for p $\in$ DestPools[SchedData ] do
4          (MinTime[SchedData,p], MinSrc[SchedData,p],
           MinDest[SchedData,p]) = FindMin (SchedData,
           Sources, p, MsgSize ) ;
5      end
6  end
7  for $d_i \in$ DataSet do
8      if $d_i$ == SchedData then continue ;
9      for p $\in$ DestPools[$d_i$] do
10         if MinSrc[$d_i$,p] == SchedSrc || MinDest[$d_i$,p]
           == SchedDest || ( mode == HalfDuplex && (
           MinSrc[$d_i$,p] == SchedDest || MinDest[$d_i$,p] ==
           SchedSrc ) ) then
11             (MinTime[$d_i$,p], MinSrc[$d_i$,p], MinDest[$d_i$,p]
               ) = FindMin ( $d_i$, Sources, p, MsgSize ) ;
12         end
13     end
14 end
15 return(MinTime, MinSrc, MinDest) ;
```

Figure 6: UpdateAfterSched()

```
1  Algorithm:Min$^3$-Allgather
   input  : MsgSize, Pool, DataSet, InitSource, mode
   output: SchedFiles
2  (SchedFiles, Sources) = ScheduleInputPool( MsgSize,
   Pool, InitSource, DataSet, mode) ;
3  if NoSubPools(Pool) $\neq$ NoHosts(Pool) then
4      for SubPool $\in$ Pool do
5          for $d_i \in$ DataSet do
6              for s $\in$ Sources[$d_i$] do
7                  if s $\in$ SubPool then
8                      SubPoolSources[$d_i$]=s ;
9                      break ;
10                 end
11             end
12         end
13         Min$^3$-Allgather(MsgSize, SubPool, DataSet,
           SubPoolSources, mode) ;
14     end
15 end
```

Figure 7: Min^3-Allgather

6. EXPERIMENTS AND RESULTS

In this Section, we compare the performance of our Min^3-Allgather strategy with various existing strategies. These existing strategies include generic allgather algorithms for homogeneous networks including MPICH algorithm [17] and Spreading Simple [8], network topology-aware methods for grids by MagPIe [14] and MPICH-G2 [18] and strategies that obtain allgather schedule for N nodes by combining N broadcast trees corresponding to N distinct root nodes. For obtaining allgather schedule from individual broadcasts, we use broadcast trees generated by Mateescu [16] and ClusteredSA [9] algorithms.

We first describe our methodology of obtaining allgather schedule for N nodes given N broadcast trees with N distinct roots. We then describe a real Grid setup involving 3-sites that we used for our allgather experiments. We then compare our Min^3-Allgather algorithm with the other strategies.

6.1 Allgather from Broadcast Trees

In this strategy, broadcast trees t_i corresponding to broadcast of data d_i by each process p_i, participating in allgather are generated. To implement allgather, each process sends and receives data on the basis of the generated broadcast trees. For constructing an allgather schedule of N nodes from N broadcast trees, a process p_i can first post N-1 non-blocking receives. p_i can then send its data d_i to a set of processes that are direct descendants of p_i in broadcast tree t_i. We denote this set of direct descendants as DirectDescendants(t_i, p_i). p_i, on receiving data d_k corresponding to a posted non-blocking receive, can send d_k to processes in DirectDescendants(t_k, p_i). However, this implementation results in poor performance, on ch_p4 device for MPICH. This is because the underlying behavior of MPICH gives better performance if sends and receives are posted in the order of their actual occurrence. This is corroborated in the work by Benson et. al. [3].

To achieve better performance for allgather constructed from broadcast trees, we use a min based algorithm for determining the order of sends at each process. The algorithm produces as output a set of files containing the order of communications of different data segments for each process. During allgather, the processes read from these files and perform the communications in the specified order. For calculating the transfer times used in the algorithm, we utilized either Single Port Half Duplex or Single Port Full Duplex model for communications.

6.2 3-Site Grid

In order to evaluate the efficiency of various strategies for allgather on grids, we utilized a grid consisting of 3 sites: 1. University of Tennessee (UT), Tennessee, USA, 2. Queen's University, Belfast, UK and 3. Vrije Universiteit, Netherlands. Details of machine specifications in each site is provided in Table 1. The bandwidths and latencies of the links between these 3 sites were measured offline and are shown in Table 2. This grid allowed us to test the performance of algorithms under 3 different setups.

Setup 1. **WAN Links across Continents:** For these experiments, we utilized machines from all sites.

Setup 2. **WAN Links within a Continent:** For these experiments, we utilized machines from both the European Sites, i.e. UK and Netherlands.

Setup 3. **LAN links within a Cluster:** For these experiments, we utilized machines from Netherlands site.

Table 1: 3-Site Grid Setup

Location	Number of machines	Specifications
Torc cluster, University of Tennessee (UT), USA	8	GNU/Linux 2.6.8, Dual PIII 933 MHz, 512 MB RAM, 40GB Hard Drive, 100 Mbps Ethernet
Queen's University, Belfast, UK	4	GNU/Linux 2.4.20, AMD Athlon 1532 MHz, 1 GB RAM, 30GB Hard Drive, Gigabit Ethernet
DAS-2, Vrije Universiteit, Netherlands	8	GNU/Linux 2.4.21, Dual PIII 996 MHz, 1 GB RAM, 20 GB Hard Drive, 100 Mbps Fast Ethernet.

Table 2: Inter-Site Bandwidths(Mbps) and Latencies(Seconds)

	UT	UK	NTH
UT	85.19, 0.00006	1.44, 0.05	1.25, 0.05
UK	1.28, 0.05	289.33, 0.000006	4.75, 0.076
NTH	1.16, 0.05	4.75, 0.076	81.86, 0.000006

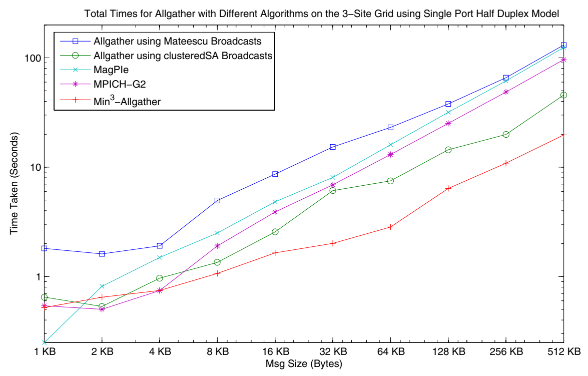
6.3 Comparison of Allgather Strategies

The total overhead in the generation of Min^3-Allgather schedules includes sorting the link bandwidths (*sort*), finding bandwidth thresholds (*thres*), formation of pools or clusters (*pools*), and determination of communication schedules (*sched*) using the Min^3-Allgather algorithm shown in Figure 7. The bandwidths on the links are determined using offline measurements and efficient mechanisms exist for the measurement and retrieval of the bandwidths [21]. Hence the bandwidth determination is not included in our Min^3-Allgather total overhead. The costs for the various overhead components in our algorithm for the 3 experiment setups are shown in Table 3. The times reported for our Min^3-Allgather in this section were obtained by adding the corresponding total overhead costs shown in Table 3 and the time taken for performing the allgather using the generated schedules.

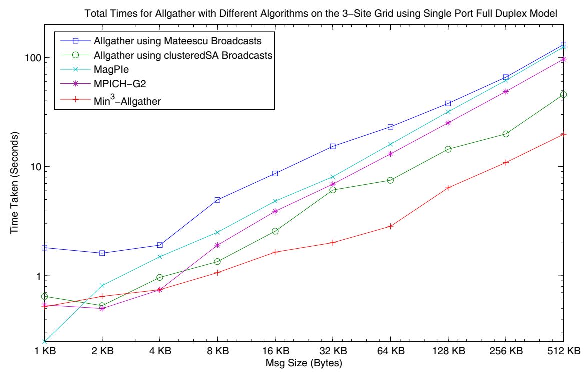
Figure 8 shows the comparison of allgather schedule gen-

Table 3: Overhead Costs (usecs.) of Min^3-Allgather

Setup	*sort*	*thres*	*pools*	*sched*	Total Overhead
Setup 1	326.86	15.71	392.29	282270	283010
Setup 2	50.7	9.3	85.8	166780	166920
Setup 3	36.75	9	19.25	108930	108990

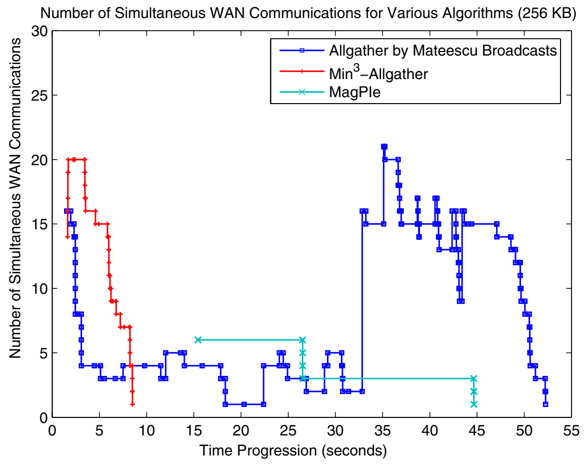


(a) Single Port Half Duplex Model

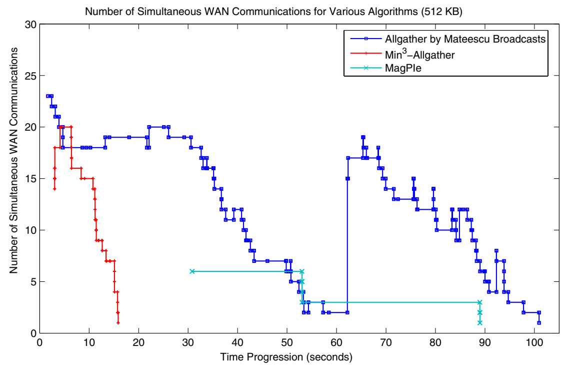


(b) Single Port Full Duplex Model

Figure 8: Allgather Results for Complete 3-Site Grid



(a) Simultaneous WAN communications for 256 KB



(b) Simultaneous WAN communications for 512 KB

Figure 9: Number of Simultaneous WAN Communications in Different Algorithms

eration strategies on the basis of average[2] actual run times for allgather for the complete grid setup on 3 sites. Except for small message sizes, the performance of Min^3-Allgather strategy is better than all other strategies. The average performance improvement of Min^3-Allgather algorithm over all the other algorithms is 42% for Half Duplex model and 52% for Full Duplex Model. We also find that the MagPIe and MPICH-G2 strategies give higher allgather times than the allgather based on our earlier developed clusteredSA broadcasts.

In order to understand the performance difference between the different algorithms for the 3-site grid setup as shown in Figure 8, we measured the number of simultaneous communications on wide-area links during the execution of allgather with a particular algorithm and message size. Figure 9 shows the number of simultaneous WAN communications during the executions of allgather with 3 different algorithms, namely, Min^3-Allgather, Mateescu's and MagPIe's, and for 2 message sizes, namely, 256 KB and 512 KB. The results correspond to using Full Duplex model for communications. The x-axis represents the time progression of allgather executions. For example, Min^3-Allgather has the lowest time to completion than the other algorithms as already seen in Figure 8. We measured the number of simultaneous WAN communications by observing the start and end times of the individual sends and receives relative to the start of the allgather. The overlap in the ranges of these start and end times gives an estimate of the number of simultaneous communications[3].

As shown in Figure 9, both Min^3-Allgather and Mateescu's strategy perform more number of simultaneous WAN communications than MagPIe's. Although allgather using Mateescu broadcasts perform more number of simultaneous WAN

[2]We take average of 4 run times

[3]Since MPICH-G2's allgather is implemented by MPI_Sendrecvs, we were not able to time the individual sends and receives and hence were not able to obtain the number of simultaneous WAN communications. However the behavior of MPICH-G2's allgather is similar to MagPIe's allgather and hence our general analysis of WAN communications in MagPIe's allgather will be applicable for MPICH-G2's allgather.

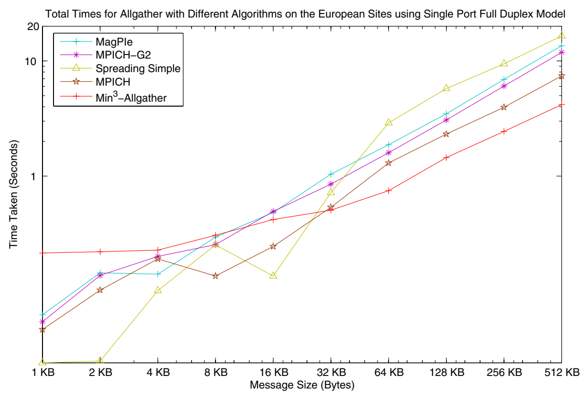


Figure 10: Allgather Results for the European Sites using Single Port Full Duplex Model

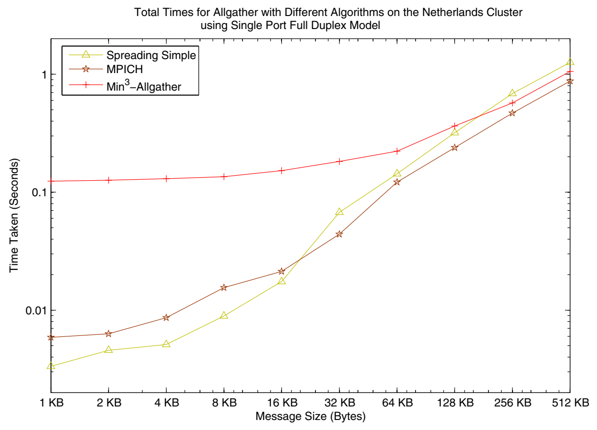


Figure 11: Allgather Results for the Netherlands Cluster using Single Port Full Duplex Model

communications than our Min^3-Allgather, the Mateescu's strategy is not wide-area optimal since it can send a data segment multiple times over a same WAN link. Hence the Mateescu's strategy has large execution times as shown in the large y-axis values in Figure 8 and large x-axis values in Figure 9. Our algorithm is effectively able to exploit the available bandwidths on WAN links and at the same time ensures wide-area optimality. We also find that WAN communications in the MagPIe's allgather strategy start later than in Min^3-Allgather and Mateescu's strategy. This is because in the initial stages of the MagPIe's allgather, local-area communications are performed to collect data at the coordinator nodes of local clusters. Only in the later stages, data is transferred between the coordinator nodes resulting in WAN communications. Thus, large percentages of executions of MagPIe and MPICH-G2 do not utilize the wide-area networks resulting in large execution times.

To evaluate the performance of our strategy on sub-parts of the 3-site Grid, we conducted experiments on a partial grid, consisting of the European machines (UK and Netherlands sites) and on a homogeneous cluster (Netherlands cluster). In both these experiments we included popular homogeneous network allgather algorithms, namely, MPICH and Spreading Simple. Figure 10 shows the comparison of different strategies for the European site. We can observe that our algorithm gives better performance than the other algorithms only for message sizes greater than 64 KB. Moreover, when compared to the results for 3 sites, the results for the 2 European sites show lesser percentage performance improvement for Min^3-Allgather over other algorithms. This is because, in the 2-site setup, Min^3-Allgather finds lesser opportunities for the formation of clusters or pools based on bandwidths and for multiple simultaneous transfers on WAN links.

Figure 11 shows the comparison of allgather strategies for a homogeneous network. Note that for this setting, we have not included MagPIe and MPICH-G2 strategies, as they are defined only for grids. For this setting, the algorithms developed for homogeneous networks perform much better. This may be attributed to the fact that though Min^3-Allgather strategy is able to utilize more bandwidth on WAN links, it does not generate best possible schedules for LAN links. Hence our Min^3-Allgather strategy is applicable to only Grid systems where the network consists of significant number of WAN links.

7. CONCLUSIONS

We have developed an algorithm for efficient allgather, a popular many to many collective communication operation, on grids. Our Min^3-Allgather algorithm is both network topology aware and network load adaptive. The algorithm follows the design principles of clustering of nodes based on transient network characteristics, parallel communications on wide-area links, and incremental construction of communication schedules. Our algorithm achieves better performance on grids, as it tries to exploit more available bandwidths on WAN links as compared to other popular approaches like MPICH-G2 [18] and MagPIe [12] and is also wide-area optimal. Experiments indicate that we achieve an average performance improvement of 52% over existing strategies.

8. FUTURE WORK

We plan to build a service-oriented architecture that constructs application-oriented allgather communication schedules similar to our previous work for broadcasts [9]. We also plan to cache popular allgather communication schedules. As network loading patterns on a Grid may be repetitive, schedules from the cache could be reused, thus saving re-computation of communication schedules. We also plan to generalize our strategy for allgather to alltoall collective communication that involves transmission of different data to different processes. This presents unique challenges in scheduling decisions as routing of data through intermediate nodes may not be beneficial.

9. REFERENCES

[1] L. B.-Estefanel and G. Mounie. Identifying Logical Homogeneous Clusters for Efficient Wide-Area Communication. In *In Proceeginds of the Euro PVM/MPI 2004*, volume LNCS Vol. 3241, pages 319–326, 2004.

[2] O. Beaumont, V. Boudet, and Y. Robert. A Realistic Model and an Efficient Heuristic for Scheduling with

Heterogenous Processors. In *Proceedings of 11th Heterogeneous Computing Workshop*, 2002.

[3] G. Benson, C.-W. Chu, Q. Huang, and S. Caglar. *A Comparison of MPICH Allgather Algorithms on Switched Networks*, volume 2840/2003 of *Lecture Notes in Computer Science*, pages 335–343. Springer Berlin / Heidelberg, September 2003. Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users' Group Meeting.

[4] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiportmessage-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.

[5] H. Casanova. Network Modeling Issues for Grid Application Scheduling. *International Journal of Foundations of Computer Science (IJFCS)*, 16(2):145–162, 2005.

[6] E. Chan, R. van de Geijn, W. Gropp, and R. Thakur. Collective Communication on Architectures that Support Simultaneous Communication over Multiple Links. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–11, 2006.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the fourth ACM SIGPLAN symposium on principles and practice of parallel programming languages (PPoPP)*, pages 1–12, 1993.

[8] A. Faraj and X. Yuan. Automatic Generation and Tuning of MPI Collective Communication Routines. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402, 2005.

[9] R. Gupta and S. Vadhiyar. Application-Oriented Adaptive MPI_Bcast for Grids. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS'06)*, Rhodes Island, Greece, 2006.

[10] L. Hollermann, T.-S. Hsu, D. Lopez, and K. Vertanen. Scheduling Problems in a Practial Allocation Model. *Journal of Combinatorial Optimization*, 1(2):129–149, 1997.

[11] N. Karonis, B. de Supinski, I. Foster, and W. Gropp. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, pages 377–386, 2000.

[12] T. Kielmann, H. Bal, S. Gorlatch, K. Verstoep, and R. Hofman. Network Performance-aware Collective Communication for Clustered Wide-area Systems. *Parallel Computing*, 27(11):1431–1456, 2001.

[13] T. Kielmann, H. Bal, and K. Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. In *IPDPS Workshops, volume 1800 of Lecture Notes in Computer Science*, pages 1176–1183, Cancun,Mexico, 2000.

[14] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140, 1999.

[15] B. Lowekamp. Discovery and Application of Network Information. PhD Thesis CMU-CS-00-147, Carnegie Mellon University, 2000.

[16] G. Mateescu. A Method for MPI Broadcast in Computational Grids. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 13*, page 251, Colorado, USA, 2005.

[17] Mpich2 home page. `http://www-unix.mcs.anl.gov/mpi/mpich2`.

[18] MPICH-G2. `http://www3.niu.edu/mpi`.

[19] K. Park, H. Lee, Y. Lee, O. Kwon, S. Park, and S. K. H.W. Park. An Efficient Collective Communication Method for Grid Scale Networks. In *Proceedings of the International Conference on Computational Science*, pages 819–828, Melbourne, Australia and St. Petersburg, Russia, June 2003.

[20] H. Saito, K. Taura, and T. Chikayama. Collective Operations for Wide-Area Message Passing Systems using Adaptive Spanning Trees. In *Proceedings of The 6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[21] M. Swany and R. Wolski. Building Performance Topologies for Computational Grids. *International Journal of High Performance Computing Applications*, 18(2):255–265, 2004.

[22] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, Spring 2005.