# A Scalable Non-blocking Multicast Scheme for Distributed DAG Scheduling*

Fengguang Song[1], Jack Dongarra[1,2], and Shirley Moore[1]

[1] University of Tennessee, EECS Department
Knoxville, Tennessee, USA
{song,dongarra,shirley}@eecs.utk.edu
[2] Oak Ridge National Laboratory,
Oak Ridge, Tennessee, USA

**Abstract.** This paper presents an application-level non-blocking multicast scheme for dynamic DAG scheduling on large-scale distributed-memory systems. The multicast scheme takes into account both network topology and space requirement of routing tables to achieve scalability. Specifically, we prove that the scheme is deadlock-free and takes at most $logN$ steps to complete. The routing table chooses appropriate neighbors to store based on topology IDs and has a small space of $O(logN)$. Although built upon MPI point-to-point operations, the experimental results show that our scheme is significantly better than the simple flat-tree method and is comparable to vendor's collective MPI operations.

## 1 Introduction and Motivations

Multicore architectures are capable of running many threads simultaneously and require future parallel software be fine-grained and asynchronous [1,2,3]. An approach to developing scalable parallel software is to place fine-grained computations in a directed acyclic graph (DAG) and schedule them dynamically (data-driven or demand-driven) [4]. Although a centralized DAG scheduler works well on SMPs, it will not scale well on systems with thousands of nodes or tens of thousands of cores. One way to overcome the scalability problem is to adopt a decentralized scheduler, that is, each node runs a private DAG scheduler and communicates with other nodes regarding data dependences only when necessary. Ideally, the distributed scheduler has no globally shared data structures, no requirement of much space to store DAGs, and no blocking operations. Furthermore, it respects the critical path, takes into account data locality, maintains load balancing, and performs communication efficiently.

Instead of solving all the problems at once, we study how to perform communication efficiently during dynamic DAG scheduling. The most common communication operation used to execute DAGs is multicasting where a completed task must notify its descendants that are blocked awaiting its output (Fig. 1).

* This material is based upon work supported by the Department of Energy Office of Science under grant No. DE-FC02-06ER25761 and by Microsoft Research.

G. Allen et al. (Eds.): ICCS 2009, Part I, LNCS 5544, pp. 195–204, 2009.
© Springer-Verlag Berlin Heidelberg 2009

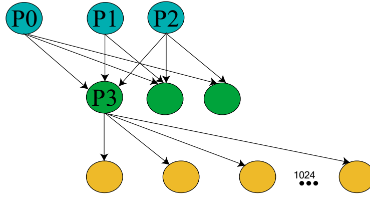



**Fig. 1.** Data multicast from parent to children in a DAG

MPI libraries provide users with optimized broadcast operations on nearly all high performance machines. Due to the dynamic and irregular parent/children relationship in general DAGs, there could be $2^N$ possible subsets of processes for broadcasting, where $N$ is the number of processes. For every finished task and its corresponding children, one has to call `MPI_Comm_create()` followed by `MPI_Bcast` to realize the multicast. Even if we ignore the time to create the communication groups, multiple MPI broadcasts involving the same process have to be executed in sequence because MPI broadcast is a collective operation. Figure 1 shows an example where P3 is involved in three communication groups with broadcast roots $P0$, $P1$, $P2$, respectively.

This paper presents a novel multicast scheme to enable dynamic DAG scheduling on large-scale distributed systems with tens of thousands of processors. The multicast scheme is non-blocking, topology-aware, scalable, and deadlock-free, and it supports multiple concurrent multicasts. We compare its performance to a flat-tree multicast and a vendor `MPI_Bcast`. Based on the experimental results, our multicasting scheme is significantly better than the simple flat-tree method and comparable to the optimized collective MPI broadcast.

## 2 Computation Model

### 2.1 Symbolic Task Graph

We represent the semantics of programs with loop nest control structures by polyhedrons such that each task instance corresponds to a unique *coordinate* or *iteration vector*. A task instance is denoted by a tuple `(type, iteration vector)`. By identifying data dependences between tasks, we are able to construct the whole DAG. Similar to the method introduced by [5], we define a task graph symbolically as follows: $G = \langle T, E \rangle$, where

$$T = \{\text{task t } : t = (type, \boldsymbol{u})\}.$$

The set of edges are defined by a set $F$ of symbolic functions:

$$F = \{f_i \text{ for certain task type } i\} \text{ and } f_i : \{\boldsymbol{u}\} \rightarrow T.$$

Given a task instance $(t, \boldsymbol{u})$, $f_t(\boldsymbol{u})$ generates a set of tasks that are dependent on task $(t, \boldsymbol{u})$.

### 2.2 Programming Model

We designed a simple application programming interface (API) and implemented a runtime system prototype to support dynamic DAG scheduling. After the user implements the API routines, the underlying runtime system can automatically parallelize and execute the DAG on shared- or distributed-memory systems. The ANSI C programming interface routines are listed below:

```
int  get_children(const Task t, Task children);
int  get_num_parents(const Task t);
void set_entry_task(const Task t);
void set_exit_task(const Task t);
```

Note that we can obtain the child tasks easily by calling `get_children()` if the finished parent task is given. As long as each member of the multicast group is notified of the parent task, it is able to deduce the whole group immediately. If the `get_children()` function is not feasible, the group members have to be included explicitly in messages.

## 3 Multicast Scheme Overview

When a set of processes are executing a DAG, multiple sources may want to notify different groups of children simultaneously. The new multicast method is able to provide this functionality automatically. The multicast scheme is essentially an application-level routing method. Every process owns a compact routing table. Although each process only has knowledge of a few neighbors, the whole group of processes is represented by a collection of hierarchical trees. Most importantly, every process is the root node of its own multicast tree. The routing algorithm simply follows the tree to multicast data to a set of children.

In Fig. 2, the process on node `001` wants to multicast data to $\{010, 100, 101\}$. For this system with eight nodes, it takes three steps to complete the multicast. There could be at most eight processes running (at leaf nodes) on the system, but certain processes will be mapped to serve as "virtual masters" responsible for their corresponding subtrees. Our method to build the routing table guarantees that there exists a path from the source to every destination by filling in "virtual masters" (Lemma 1 in Sect. 7). The path length is bounded by $logN$.

## 4 Topology ID

To improve communication performance, it is critical to know the communication cost between a node and the other nodes on a system. One could build a $N \times N$ table to describe the latency and bandwidth information between every pair of nodes. But for a system with millions of nodes, it is too costly to build and maintain such a big table. Another natural approach is to use hierarchy as an abstraction to achieve scalability. The hierarchy abstraction has been widely used on the Internet, for example for DNS and IP addresses, as well as for message passing operations

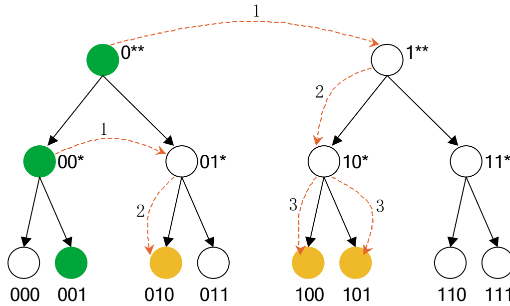



**Fig. 2.** Data multicasting in a data flow graph

on computational Grids. Instead of exposing all the other nodes to the source, the hierarchy technique utilizes a hierarchical tree to send data level by level. This way each node only communicates with a small number of nodes so that the system keeps scaling. We assign each node a topology ID on the system. Working like ZIP codes, we assume the longer the common prefix of the two nodes' topology IDs, the closer they are and the smaller the latency. When a process is running on a node with topology ID $x$, we say the process has topology ID $x$.

## 5 Extention to the Plaxton Neighbor Table

In this section, we briefly describe the Plaxton neighbor table and our extension to support multicasting. Plaxton uses an incremental routing approach similar to hypercube routing which resolves the destination node address dimension by dimension [6]. Supposes a system has $n = 2^m$ nodes, where $m$ is a multiple of $b$. Plaxton assumes that each node has a label which is independently and uniformly distributed at random between 0 and $n-1$. Instead of using a random label for each node, we assign a topology ID $tid \in [0...n-1]$ to each node. The topology ID reflects the latency relationship (near or far) between two nodes, and is expressed as a sequence of $\frac{m}{b}$ digits with the base $2^b$. For instance, if one system has $4096 = 2^{12}$ nodes, base $= 2^3$ leads to a 4-digit octal topology ID.

Every node has its own neighbor table $T$. Table $T$ consists of $r$ rows and $c$ columns, where $r$ is equal to the number of digits $(=\frac{m}{b})$ and $c$ is equal to the digit's base $(= 2^b)$. Table entry $T[i, j]$ stores the forwarding node address. In the case of application-level multicasting, we store an MPI rank as a forwarding address. Let node $x$ have a topology ID of $id^{(x)} = d_0 d_1 \ldots d_{r-1}$. If table entry $T[i, j]$ in node $x$ contains node $y$ which has topology ID $id^{(y)}$, then $id^{(x)}$ and $id^{(y)}$ must satisfy the following two conditions:

(1) $id_0^{(x)} id_1^{(x)} \ldots id_{i-1}^{(x)} = id_0^{(y)} id_1^{(y)} \ldots id_{i-1}^{(y)} = d_0 d_1 \ldots d_{i-1}$,
(2) $id_i^{(x)} \neq j$ and $id_i^{(y)} = j$.

Please note that there could exist a set $Y$ of nodes meeting the above conditions for node $x$. For instance, table entry $T[3, 5]$ in a node with octal topology ID

012345 can contain any node with a topology ID $\in$ $012[\{0-7\}/3\}][0-7]^{+}$. Therefore a decision function is needed to choose the best candidate. For instance, Plaxton chooses $y^{*} = Min_{y \in Y} CommCost(x, y)$ as the best neighbor.

In the case of $Y = \emptyset$, Plaxton assumes an ordering in the set of $n$ nodes and picks a node $y$ that matches node $x$ in the suffix $i, i+1, \ldots, r-1$ digits with the highest order. In contrast to the full Plaxton neighbor table, we leave those entries empty and prove this modification avoids cycles in application-level multicasting (Theorem 2 in Sect. 7).

### 5.1 Compact Routing Table

While routing tables are usually used to connect nodes, we use them to connect processors and processes in the context of application-level multicasting. Assume a system has $n$ processors and the base of topology IDs is equal to $c$, then the routing table will have $\frac{log_2(n)}{log_2(c)}$ rows and $c$ columns. The routing table occupies a small amount of space even for large-scale systems. If a system has one million ($2^{20}$) cores, a base of 16 results in a routing table of 5 rows by 16 columns that equals 80 entries. If one has a billion ($2^{30}$) cores, the routing table is of 6 rows and 32 columns given the base 32. Every table entry just stores a single integer.

## 6 Algorithms

This section describes how every process builds its own routing table when the application first starts and how the process constantly receives messages and forwards them to proper destinations.

### 6.1 Building a Local Routing Table

Before doing any real work, each MPI process first builds a local routing table. Each process's topology ID is assigned by users based on the network topology. In Fig. 3, a process scans every other process ID and compares that process's topology ID to its own topology ID to fill in the routing table. When there are multiple processes that are legitimate to be stored in T[i,j], we either pick a process randomly or find the closest process. In our experiments on Myrinet, the random method is slightly better than the nearest neighbor method.

### 6.2 Forwarding Algorithm

While participating in the multicast, a process works as either an internal node or a leaf in the multicast tree. Whenever the root is given, the locations of receivers become fixed in the particular multicast tree. Data will always flow from root to leaves. Figure 3 shows how to find the next level of tree nodes in the multicast tree to which to forward. The index of the next level (i.e., `stage`) should be at least one level further from the root. The program looks up the table and gets a forwarding process for each child and stores it in array `destinations`.

```
typedef struct {
  int  table[NUM_LEVELS * NUM_COLS];
  int  topo_ids[MAX_NUM_PROCESSES];
}* NeighborTable;

int *candidates[NUM_LEVELS * NUM_COLS];
NeighborTable my_tbl;

for(p = 0; p < nprocs; p++) {
  if(p  == my_pid) continue;
  topid  = my_tbl->topo_ids[p];
  level  = longest_prefix(my_top_id, topid);
  column = get_kth_digit(topid, level);
  idx    = level * NUM_COLS + column;
  candidates[idx][counters[idx]++] = p;
}

/*choose proper neighbor from candidates*/
choose_best_neighbor(my_tbl, candidates);
```

```
while(1) {
  ...
  Received a message from process prev_topid;
  stage = longest_prefix(my_top_id, prev_topid)+1;
  for(i = 0; i < num_children; i++) {
    p = get_children(i);
    if(p == my_pid) continue;
    top_id = my_tbl->topo_ids[p];
    lcl = longest_prefix(my_top_id, top_id);
    if( lcl  >= stage) {
      column  = get_kth_digit(top_id, lcl);
      forward = TBL_ENTRY(my_tbl, lcl, column);
      if(!is_element(forward, destinations)) {
        destinations[idx++] = forward;
      }
    }
  }
  Send message to processes in destinations[];
}
```

**Fig. 3.** Algorithms for the non-blocking multicast scheme

## 7 Theorems

**Lemma 1.** *Suppose process $P_x$ has a topology ID $x$ and needs to send data to process $P_z$ with topology ID $z$. Then there always exists a process $P_y$ stored in $P_x$'s neighbor table such that $P_x$ can forward data to $P_y$ and $LCD(y, z) \geq LCD(x, z) + 1$.*

*Proof.* Let $LCD(i, j)$ compute the longest common prefix length of $i$ and $j$. Suppose $i = LCD(x, z)$, $P_x$ will forward data to a process with a topology ID of the form $x_0x_1 \ldots x_{i-1}z_i * \ldots *$. It is easy to see that at least $z$ have the form. So one of the processes of $P_z \cup \{\text{processes with ID } x_0x_1 \ldots x_{i-1}z_i * \ldots *\}$ will get the forwarded data. Therefore such a process must exist. By definition of $LCD$, we know $x_0x_1 \ldots x_{i-1} = z_0z_1 \ldots z_{i-1}$ and $x_i \neq z_i$. Given $i$ and $z$, the forwaring algorithm chooses the process stored in the $i$th row and the $z_i$th column. Any process located in $T[i, z_i]$ will be the target to which $P_x$ forwards data and it must exist. WLOG, let it be $P_y$ with topology ID $y$. Since $P_y$ is in $T[i, z_i]$ of $P_x$'s neighbor table, $y_0y_1 \ldots y_{i-1} = x_0x_1 \ldots x_{i-1} = z_0z_1 \ldots z_{i-1}$ and $y_i = z_i$. Therefore, $y_0y_1 \ldots y_i = z_0z_1 \ldots z_i$. In other words, $LCD(y, z) = i + 1 \geq LCD(x, z) + 1$.

Lemma 1 proves that the forwarding method is always successful even if there exist empty entries in the process's routing table. For every step of forwarding, the longest common prefix length to the destination increases by at least one.

**Theorem 1 (Reachability).** *It is always possible to route a message from process $P_x$ to process $P_z$ and it takes at most $m$ steps to reach $P_z$. $m$ is the number of digits in topology IDs.*

*Proof.* By Lemma 1, there $\exists P_y$ such that $P_x$ can forward data to $P_y$ and $LCD(y, z) \geq LCD(x, z) + 1$. Since topology ID $z$ has $m$ digits, it takes at most $m$ steps to send data to $P_z$.

**Theorem 2 (Deadlock-freedom).** *The forwarding mechanism guarantees that there is no cycle during the forwarding process.*

*Proof.* Suppose process $x_1$ wants to send a message to $x_p$, but there is a cycle $x_1 \rightarrow x_2 \ldots \rightarrow x_k \rightarrow x_1$ formed before the message reaches $x_p$. If $LCD(x_1, x_p) = d_0 d_1 \ldots d_{l_1}$, then by Lemma 1,

$$LCD(x_2, x_p) = d_0 d_1 \ldots d_{l_1} \ldots d_{l_2}$$

$$LCD(x_3, x_p) = d_0 d_1 \ldots d_{l_1} \ldots d_{l_2} \ldots d_{l_3}$$

$$\ldots$$

$$LCD(x_k, x_p) = d_0 d_1 \ldots d_{l_1} \ldots d_{l_2} \ldots d_{l_3} \ldots d_{l_k}$$

$$LCD(x_1, x_p) = d_0 d_1 \ldots d_{l_1} \ldots d_{l_2} \ldots d_{l_3} \ldots d_{l_k} \ldots d_{l_{k+1}}$$

There is a contradiction if we compare the first $LCD(x_1, x_p)$ and the last $LCD(x_1, x_p)$. Therefore, the forwarding mechanism guarantees there is no cycle.

## 8 Related Work

MPICH-G2 uses depth to represent where an MPI process is located in a computational Grid [7]. The depths include levels of *wide area*, *local area*, *system area*, and *machine-specific area*. The topology table represented by *depths* and *colors* is a global table for the whole grid and needs to be accessible by every process. Our topology ID representation has a distributed compact table and requires much less space.

Plaxton introduces local neighbor tables at each node [6]. Our work is an extension to Plaxton's neighbor table where we build routing tables for MPI processes instead of nodes. Since a user's processes are always a subset of all the nodes on the system, we modify the table-building method to allow empty entries (or "holes") in the routing table. In addition, we design a multicast scheme based on the extended routing table. Wu [8] designs a deadlock-free prefix-based multicasting scheme for irregular networks. Each outgoing channel of a node is assigned a label. The multicast packet is first forwarded up to the root and then forwarded down to leaves. But the whole system is based on a single spanning tree. In our multicast method, every process has its own spanning tree. Panda proposes a Hierarchical Leader Based approach to support one-to-many multicasting [9]. The set of nodes are grouped into subsets explicitly so that each subset is represented by a leader. Banerjee et al. also uses a hierarchical clustering method to multicast the data stream to large receiver sets [10]. In contrast, our method builds routing tables to form hierarchies automatically (represented by spanning trees).

Bayeux uses the structure of Tapestry to provide an application-level multicast scheme for streaming multimedia applications [11,12]. Bayeux builds a distribution tree based on four control messages: *JOIN*, *LEAVE*, *TREE*, *PRUNE*. To construct a distribution tree, the source server must advertise the session information first. Then the clients have to join the session to form a tree. We don't

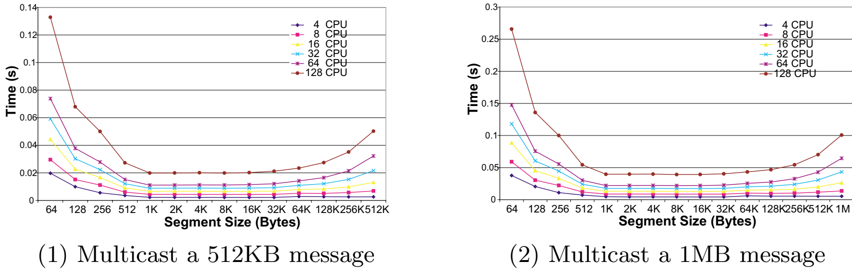
need to construct distribution trees and simply use the implicit spanning trees to multicast data.

# 9 Experiments

We conducted experiments on a cluster machine with 64 nodes each with two processors. The cluster is connected by a Myrinet network. We also did experiments on a SGI Altix 3700 BX2 machine which has a fat tree network topology. The performance result on the SGI machine is similar to that on the cluster and is not shown here due to the space limitation.

## 9.1 Effect of Segments

The performance of the non-blocking multicast method could be affected by the segment size. Given a message size, we can choose to send it out once or in a number of segments. Figure 4 considers two message sizes: 512KB and 1MB. For each message size, we use different segments with sizes from 64Bytes to the whole message size and run it on a range of processors from 4 CPUs to 128 CPUs. Based on the data from Fig. 4, a segment size between 1KB and 32 KB always produces the best performance. Therefore in the experiments described in Sect. 9.2, we choose the segment size 8KB for our multicast method whenever possible.



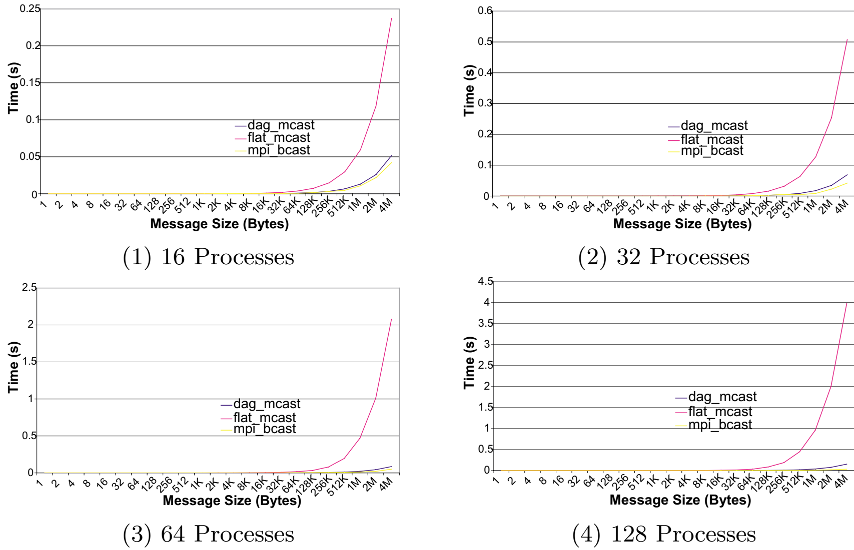

(1) Multicast a 512KB message (2) Multicast a 1MB message

**Fig. 4.** Performance of multicast varies with different segment size on Myrinet

## 9.2 Experimental Results

We compare our non-blocking multicast method (labeled as "dag_mcast") to Myricom's MPICH-MX 1.1 MPI_Bcast (labeled as "mpi_bcast") and a straightforward implementation that uses a flat-tree to perform multicasting (labeled as "flat_mcast"). The flat-tree method simply sends the message to every destination one by one. Both flat_mcast and dag_mcast are implemented using point-to-point `MPI_Send` and `MPI_Recv` operations.

We conducted experiments on a range of processors from 16 up to 128. From Fig. 5, we can see that both `dag_mcast` and `mpi_bcast` are significantly better




(1) 16 Processes (2) 32 Processes

(3) 64 Processes (4) 128 Processes

**Fig. 5.** Multicast performance on a cluster connected with Myrinet

than `flat_mcast`. And the non-blocking multicast method is comparable to the highly-optimized collective MPI_Bcast. Note that the time to invoke `MPI_Init` and `MPI_Comm_create` was not counted for the `mpi_bcast` experiments (in favor of `mpi_bcast`). The reason why the non-blocking multicast method is slower than MPI_Bcast is because our implementation is built over MPI point-to-point operations and we cannot do similar optimizations as MPI collective operations do (e.g., broadcast may be implemented as `scatter` followed by `allgather`, optimal binomial tree is built in advance). Although MPI_Bcast is faster, it is difficult to create communication groups and do collective broadcasts for every distinct group in dynamic DAG scheduling programs.

## 10 Conclusion

Our non-blocking multicast scheme is designed to support dynamic DAG scheduling on distributed-memory machines. While it is possible to use `MPI_Bcast` directly to implement it, creating communication groups and performing collective operations for arbitrary sets of parent/children is cumbersome to program. We have designed a multicast scheme, using topology IDs, compact routing tables, and multiple spanning trees. The multicast scheme is proven to be deadlock free, scalable in terms of time and space, topology-aware, and non-blocking. Our experimental results show that performance of our scheme is significantly better than the simple flat-tree method and comparable to vendor-optimized collective MPI operations.

## References


1. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M.: Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph. 27(3), 1–15 (2008)
2. Golla, R.: Niagara2: A highly threaded server-on-a-chip (2007)
3. Le, H.Q., Starke, W.J., Fields, J.S., O'Connell, F.P., Nguyen, D.Q., Ronchetti, B.J., Sauer, W.M., Schwarz, E.M., Vaden, M.T.: IBM Power6 microarchitecture. IBM J. Res. Dev. 51(6), 639–662 (2007)
4. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Lapack working Note 191 (2007)
5. Cosnard, M., Jeannot, E.: Compact dag representation and its dynamic scheduling. J. Parallel Distrib. Comput. 58(3), 487–514 (1999)
6. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing nearby copies of replicated objects in a distributed environment. In: SPAA 1997: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures, pp. 311–320. ACM Press, New York (1997)
7. Karonis, N.T., Toonen, B., Foster, I.: MPICH-G2: A Grid-enabled implementation of the message passing interface. Journal of Parallel and Distributed Computing 63(5), 551–563 (2003); Special Issue on Computational Grids
8. Wu, J., Sheng, L.: Deadlock-free multicasting in irregular networks using prefix routing. The Journal of Supercomputing 31, 63–78 (2005)
9. Panda, D., Singal, S., Kesavan, R.: Multidestination message passing in wormhole k-ary n-cube networks with base routing conformed paths. IEEE Transactions on Parallel and Distributed Systems 10(1), 76–96 (1999)
10. Banerjee, S., Bhattacharjee, B., Kommareddy, C.: Scalable application layer multicast. In: SIGCOMM 2002: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 205–217. ACM, New York (2002)
11. Zhao, B.Y., Kubiatowicz, J.D., Joseph, A.D.: Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, Berkeley, CA, USA (2001)
12. Zhuang, S.Q., Zhao, B.Y., Joseph, A.D., Katz, R.H., Kubiatowicz, J.D.: Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In: NOSSDAV 2001: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video, pp. 11–20. ACM Press, New York (2001)