In-Advance Path Reservation For File Transfers In e-Science Applications

Yan Li, Sanjay Ranka and Sartaj Sahni
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, Florida 32611
Email: {yanli, ranka, sahni}@cise.ufl.edu

***Abstract*—We develop multi-path reservation algorithms for in-advance scheduling of large file transfers in connection-oriented optical networks. Specifically, to schedule a single file transfer to complete at the earliest possible time, a new max-flow based greedy algorithm(GOS) and four variants that adapt the k-shortest paths and k-disjoint paths algorithms are proposed. Meanwhile, to find an earliest-finishing schedule for a batch of file transfers, a linear programming based algorithm($BATCH$) is developed. Extensive experiments using both real world and random networks show that our GOS algorithm provides a good balance among maximum finish time, average finish time, and computational complexity. Although our $BATCH$ algorithm results in the smallest maximum finish time, this algorithm has a significantly larger computational requirement than our other algorithms. GOS yields file transfer schedules with similar maximum finish time and reduces average finish time while having a significantly less computational requirement.**

I. Introduction

The rapid development of high speed optical networks has enabled a variety of e-Science applications that are both data intensive and geographically distributed. These applications, which include data mining, data consolidation and alignment, storage, visualization and analysis [1], generate large amounts of data (order of terabytes to petabytes) and require these data to be transferred across the network. For example, simulation data sets produced by a supercomputer may need to be archived in a remote storage center and later analyzed by a different supercomputer that is located at a third site. When the overall job run time is considered, the file transfer time may become a major bottleneck because this time may become unbounded due to network congestion and failures. Thus, dedicated connections, especially dedicated bandwidth channels, are essential to offer (i) large capacities for massive data transfer operations, and (ii) dynamically stable bandwidth for monitoring and steering operations. The importance of dedicated connection capabilities has been recognized, and there are several ongoing network research projects that develop such capabilities [2], [3]. In addition, production networks at the national and international scale with such capabilities are being deployed in Internet2 [4]. These networks usually have a sufficiently small-sized backbone that it is practical to employ centralized management on the network's bandwidth, including user requests scheduling, path identification and bandwidth allocation.

Consider a scenario where a large number of files have to be transferred from multiple sources to multiple destinations. Each file corresponds to a transmission between a single pair of nodes. The objective is to minimize the transfer time of all files across the dedicated optical network, which we call *Earliest Finish Time File Transfer Problem* (EFTFTP). In this paper, we develop and evaluate algorithms to solve this problem. We consider two fundamentally different approaches to schedule file transfers: (i) **Online Scheduling (EFT-Online)** in which file transfer requests are scheduled one by one, in the order of their arrival without the knowledge of any future requests and (ii) **Periodic Batch Scheduling (EFT-Batch)** in which requests are collected/batched in a centralized scheduler; the collected/batched requests are scheduled as a group with certain periodicity. The periodicity may be defined in terms of time (say, every 15 minutes), or number of batched requests (say, whenever 10 file transfer requests have been batched) or a combination of these two metrics (say, the the smaller of 15 minutes and the time required to batch 10 transfer requests), and so on. Online scheduling is a special case of batch scheduling if scheduling is done whenever one job arrives (batch size=1). Our simulations show that our GOS algorithm results in a slightly larger finish time than does periodic batch scheduling. However, our GOS algorithm requires significantly less computation time and has better performance with respect to the mean finish time of the file transfers.

The rest of the paper is organized as follows. In Section II, we describe related work. Our network model and terminology are detailed in III. In Section IV, we present the greedy algorithm for online scheduling and prove that this algorithm minimizes the finish time of the current file transfer being scheduled. Four variants of this greedy online algorithm are also proposed in Section IV. These variants aim to reduce computation time with limited increment in finish time. A linear programming formulation for periodic batch scheduling is developed in Section V. This formulation minimizes the finish time of the entire request pack, i.e. the last completed file transfer of the batch. In Section VI, we evaluate our online and batch algorithms together with the k-shortest paths algorithm of [5] and the k-disjoint paths algorithm of [6]. Finally, we conclude in Section VII.

II. Related Work

Generally, bandwidth reservation systems operate in one of two modes [7]:

(a) In **on-demand** scheduling, bandwidth is reserved for a time period that begins at the current time.
(b) In **in-advance** scheduling, bandwidth is reserved for a time period that begins at some future time.

On-demand scheduling is a special case of in-advance scheduling: the time interval between the request's arrival and its actual starting time is zero. On-demand scheduling, which is typically supported by Multiple Protocol Label Switching (MPLS) [8] at layer 3 and by Generalized MPLS (GMPLS) [9] at layers 1 and 2, is supported by CHEETAH, DRAGON, and UCLP. Typical algorithms for on-demand scheduling can be found in [10], [11]. In-advance scheduling is supported by GeantII, OSCARS, USN and Enlightened network. The corresponding algorithms are described in [12]–[15].

Although much of the research on bandwidth scheduling has focused on reserving a single simple path for a specified bandwidth request, it is well known that using multiple paths improves the utilization of the available network resources [16]. The multi-path reservation problem is formulated in [16] as a network flow problem with the objective of minimizing link congestion. Algorithms for delay-constrained file transfer using multiple paths are proposed in [17]. The multi-path file transfer scheme is considered with both link utilization constraints and path length constraints in [18]. A maximum concurrent flow formulation is used in [6] to solve the large file transfer problem with fixed start and end times; the objective is to maximize network throughput. [18] also develops linear programming models to maximize the network throughput and proposes two heuristics for multi-path routing. The first heuristic, k-Shortest Paths (KSP), uses the *k-shortest paths* algorithm of [5] to compute k not necessarily disjoint paths from the source to the destination. The scheduling of the file transfer is restricted to these k paths. The second heuristic, k-Disjoint Paths (KDP), computes k disjoint paths from source to destination by eliminating the links contained in previously computed paths before computing the next path; each path computation generates the shortest path in the remaining network. Experimental results reported in [6] indicate that these heuristics result in a network throughput similar to what is obtained from the linear programming formulation albeit at a much reduced computation cost. In Section IV, these two heuristics are used by us to develop variants of our GOS algorithm.

III. Network Model and Terminology

We assume that the network is represented as a directed graph $G = (V, E)$. Each node of this graph represents a device such as a switch for layers 1-2 and a router for layer 3; and each edge represents a link such as SONET or Ethernet. Each link has a *rated capacity*, which is the maximum number of bytes of data that may flow through the link per second. We assume that the rated capacity of each link of G is more than 0. At any specific instance, the available bandwidth on a link may be less than its rated capacity because of pre-scheduled traffic. When developing a bandwidth reservation system, one must decide on the representation of time. The options are to either consider time as divided into equal size slots as is done in [12], [13] or to consider time as being continuous as in [1], [14]. For space efficiency and accuracy as explained in [7], we use the continuous time model in this paper. In this model, the status of each link l is maintained using a time-bandwidth list (TB list) $TB[l]$ that is comprised of tuples of the form (t_i, b_i), where t_i is a time and b_i is a bandwidth. The tuples in a TB list are in increasing order of t_i. If (t_i, b_i) is a tuple of $TB[l]$ (other than the last one), then the bandwidth available on link l from t_i to t_{i+1} is b_i. When (t_i, b_i) is the last tuple, a bandwidth of b_i is available from t_i to ∞. Each TB list can be represented as an array using dynamic array resizing method as described in [19] or as a linked list.

Let $T = [T_0, T_1, \cdots]$, $T_0 < T_1 < \cdots$, be the union of the time component of the (t_i, b_i) tuples in the TB lists of all links in the network. We refer to T as the *global time list*. It is easy to see that the available bandwidth on each link of the network is unchanged in the interval $[T_i, T_{i+1})$. Figure 1 shows the TB lists for 2 links. For this simple example, assume these are the only two links in the network. The TB list for the first link is $[(0, 5), (1, 2), (2, 5)]$ and that for the second link is $[(0, 5), (1.5, 3), (2, 5)]$. The global time list for our example is $[0, 1, 1.5, 2]$. In the interval [0,1), the available bandwidth on the two links is 5 whereas in the interval [1,1.5), the first link has an available bandwidth of 2 while the second link's available bandwidth is 5, and neither of links' bandwidth changes within this basic interval.

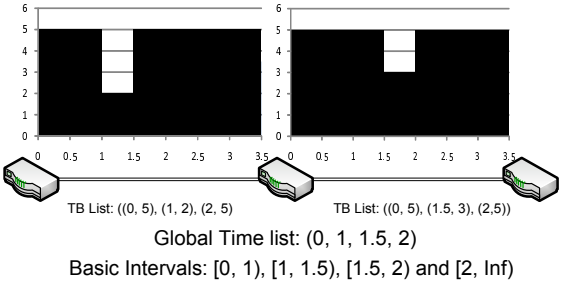


Fig. 1. Basic intervals

The intervals $[T_0, T_1)$, $[T_1, T_2)$, $\cdots$ in the global time list are referred as *basic intervals*. At any time within a certain basic interval, each edge has a constant amount of available bandwidth. Basic intervals obtained from the global time list can be ordered using the relationship $[a, b) < [c, d)$ iff $b \leq c$ (note that the basic intervals of a global time list are disjoint and that $a < b$ for each basic interval $[a, b)$).

File transfer requests are characterized by a 5-tuple $(s_i, d_i, f_i, A_i, S_i)$ where s_i is the source location of the file that is to be transferred; d_i is the destination to which the file is to be sent; f_i is the size of the file; A_i is the time when the ith file transfer request is made; and S_i, which is the time at which the file becomes available for transfer, specifies the earliest time at which the file transfer may begin. We may assume that $A_i \leq S_i$ and if $A_i = S_i$, the scheduler performs on-demand scheduling.

IV. Online Scheduling

We propose five online file transfer scheduling algorithms. The first is a greedy algorithm that employs network flows to minimize the finish time of the file transfer being scheduled. The remaining four are adaptations of this optimal greedy algorithm. These adaptations use k-shortest paths or k-disjoint paths to reduce the complexity of the scheduling algorithm, but yield little in maximum finish time.

```
GOS(i, G)
{
    Construct the current global time list T from the
        TB lists;
    Delete from T all T_i ≤ S_i;
    Insert S_i and ∞ into T and relabel the members of T in
        ascending order beginning with the label T_0;
    rfs = f_i;        //remaining file size
    j = 0;        //basic interval index;
    while (rfs > 0)
    {
        Let N be the network derived from G by assigning
            to each link a capacity equals to its available
            bandwidth in the basic interval [T_j, T_{j+1}),
        Remove the links with 0 capacity from N;
        maxFlow = Max flow from s_i to d_i in N;
        maxTime = min{T_j + rfs/maxFlow, T_{j+1}};
        size = (maxTime − T_j) * maxFlow;
        Schedule the transfer of size bytes from T_j to
            maxTime using the max flow links;
        Update the TB lists of the max flow links;
        rfs −= size;
        j++;
    }
}
```

Fig. 2. Greedy online scheduling algorithm GOS

A. Greedy Algorithm

Our greedy online algorithm, GOS, schedules a file transfer $(s_i, d_i, f_i, A_i, S_i)$ by examining the basic intervals in the network's current global time list in increasing order. The examination begins with the basic interval that includes the time S_i. In each examined interval, we transfer as much of the file as is possible. This maximum amount can be determined using a max-flow algorithm (see [20], for example). The examination of basic intervals stops when all f_i bytes of the file have been scheduled. Figure 2 gives our greedy online algorithm, GOS, to schedule the ith request. This algorithm uses a subgraph N of G comprised only of the links that have some available bandwidth in the current basic interval. We use the term *max flow links* to denote those edges of N that have a non-zero flow in the max flow solution for N. Also, since $maxFlow$ may be zero in some basic intervals, care needs to be taken when programming algorithm GOS to avoid a divide by zero error when computing $rfs/maxFlow$.

Theorem 1: If G has a path from s_i to d_i, then Algorithm GOS schedules the ith file transfer request $(s_i, d_i, f_i, A_i, S_i)$ so as to complete at the earliest possible time.

Proof: From the following facts (a) G has a path from s_i to d_i, (b) the rated capacity of each link of G is more than 0, (c) the last basic interval of the global time list always extends to ∞ , and (d) the available bandwidth of each link is its rated capacity during this last basic interval, it follows that the max flow from s_i to d_i in the last basic interval is non-zero and so the remaining file size rfs can always be scheduled for transfer in this last basic interval. Hence, GOS is able to schedule every file transfer request.

Let the finish time of a file transfer schedule constructed by GOS be ft. Note that ft is the value of $maxTime$ when GOS terminates. We show, by contradiction, that ft is the earliest possible time at which this file transfer can complete. Suppose there is another transfer schedule, S, for the same request that completes the transfer by time $ft' < ft$. Let q be such that $T_q \leq ft < T_{q+1}$ (all global time references in this proof are to times as relabeled by GOS) and let q' be such that $T_{q'} \leq ft' < T_{q'+1}$. Note that $q' \leq q$. If $q' < q$, then there is a basic interval $u < q$ such that the amount of f_i scheduled for transfer in interval u by schedule S is more than that scheduled for transfer in u by the GOS schedule. This isn't possible since the GOS schedule transfers the maximum possible amount in each basic interval prior to q. If $q' = q$, then since $ft' < ft$, the amount scheduled for transfer by S from T_q to ft' is less than that scheduled for transfer by the GOS schedule from T_q to ft, or the flow used by the GOS schedule after T_q cannot be the maximum flow. Hence, there must be a basic interval $u < q = q'$ in which more of f_i is scheduled for transfer by S than by the GOS schedule. As noted earlier, this isn't possible. Hence, there is no transfer schedule S with $ft' < ft$. ■

The complexity of algorithm GOS is determined by the complexity of the max flow algorithm that is used as well as by the number of basic intervals in the global time list. The complexity of the push-relabel max flow algorithm described in [20] is $O(n^3)$, where n is the number of vertices in the network flow graph. For networks with few edges, the sparse graph network flow algorithm of Sleator and Tarjan (see [20], for example) may be used. The complexity of this algorithm is $O(nm \log n)$, where m is the number of links in the network. When scheduling the ith file transfer, the size of the global time list is $O(i)$, since each previously scheduled request will increase the size of global time list by at most 2: job's start and end time. So, the complexity of GOS is $O(n^3 i)$ when the push-relabel max flow algorithm is used and $O(nmi \log n)$ when the sparse graph max flow algorithm is used. Since typical computer networks are generally sparse and have only $O(n)$ links, using the sparse graph max flow algorithm results in a complexity of $O(n^2 i \log n)$ for GOS.

B. Variants

We incorporate the idea behind KSP and KDP scheduling [6] into algorithm GOS so as to reduce the time it takes to schedule a file transfer. In the KSP and KDP adaptations of GOS, when scheduling the request $(s_i, d_i, f_i, A_i, S_i)$, rather than work with the entire network graph G as is done by GOS, we work with the subgraph defined by the k paths from s_i to d_i. In the case of the KDP adaptation, since the k paths are disjoint the max flow from s_i to d_i in any basic interval is easily seen to be the sum of the minimum available capacity of a link on each of the k paths. So, we avoid running a complex network flow algorithm to determine the max flow. In the case of the KSP adaptation, since the paths are not disjoint, we still need to run the GOS algorithm on the network formed by these k paths. However, since the size of the network being considered is smaller, run time is reduced.

For both the KSP and KDP adaptations we define a static and a dynamic variant. In the *static variant* the cost of a link is

defined to be its rated capacity (alternatively, some other non-changing cost may be assigned) In the *dynamic variant*, links are assigned a cost each time a scheduling request arrives and the k shortest paths to use are computed using these newly assigned link costs. The cost assigned to a link in the dynamic variant is proportional to the fraction of its rated capacity that has been committed from the current time to the finish time of the last finishing file transfer so far scheduled in the network. The static and dynamic variants of the KSP and KDP adaptations of GOS are referred to as KSP-S, KSP-D, KDP-S, and KDP-D, respectively.

V. Periodic Batch Scheduling Algorithm

In periodic batch scheduling, requests are collected/batched in a centralized scheduler and scheduled as a group with certain periodicity. The periodicity may be defined in terms of time (say, every 15 minutes), in terms of number of requests batched (say, whenever 10 file transfer requests have been batched), a combination of these two metrics (say, the earlier of 15 minutes and the time required to batch 10 transfer requests), and so on.

We develop a 2 step algorithm to optimally (i.e., minimize the maximum finish time) schedule a set of file transfer requests. The two steps are:

Step 1: Determine the minimum finish time, $minFinishTime$.

Step 2: Determine a file transfer schedule that achieves this minimum finish time.

$$\min \quad ft \tag{1}$$

$$\text{subject to} \sum_{k:(l,k)\in E} f_{lk}^j(q) - \sum_{k:(k,l)\in E} f_{kl}^j(q) = 0$$

$$\forall j \in F, \forall l \in V, l \neq s_j, l \neq d_j, 0 \leq q \leq i \tag{2}$$

$$\sum_{q=0}^{i} (\sum_{k:(l,k)\in E} f_{lk}^j(q) - \sum_{k:(k,l)\in E} f_{kl}^j(q)) =$$

$$\begin{cases} f_j & \text{if } l = s_j \\ -f_j & \text{if } l = d_j \end{cases} \quad \forall j \in F \tag{3}$$

$$\sum_{j\in F} f_{lk}^j(q) \leq b_{lk}(T_q) * (T_{q+1} - T_q), \forall (l,k) \in E, q < i \tag{4}$$

$$\sum_{j\in F} f_{lk}^j(i) \leq b_{lk}(T_i) * (ft - T_i), \forall (l,k) \in E \tag{5}$$

$$f_{lk}^j(q) \geq 0, [T_q, T_{q+1}] \subseteq [S_j, T_{q+1}], \forall (l,k) \in E, \forall j \in F \tag{6}$$

$$f_{lk}^j(q) = 0, [T_q, T_{q+1}] \nsubseteq [S_j, T_{q+1}], \forall (l,k) \in E, \forall j \in F \tag{7}$$

For the first step, we construct a global time list from the TB lists of all links as before and then construct the basic intervals from this global time list. The basic interval $[T_i, T_{i+1})$ is referred to simply as basic interval i. To determine the minimum finish time, we use a linear programming (LP) model to determine, for a specified basic interval i, the minimum time within this basic interval by which it is possible to complete all file transfers in the given request set F. This LP model will have no feasible solution for basic intervals i if it isn't possible to complete the file transfer by time T_{i+1}. In this case, $minFinishTime$ must lie in a basic interval $q > i$. Suppose the value of LP's objective function ft is a valid time within the basic interval i. Then all the jobs in the batch F can be finished by ft. Now, $T_i \leq ft \leq T_{i+1}$. If $ft > T_i$, $minFinishTime = ft$. However, when $ft = T_i$, it is possible to complete the file transfers in an interval $q < i$. So, using the LP model, we can conduct a binary search over the basic intervals to determine the value of $minFinishTime$.

Equations 1 through 7 give our LP model to find $minFinishTime$ within $[T_i, T_{i+1})$. In this formulation, $ft \in [T_i, T_{i+1})$ denotes the time by which all file transfers complete. $f_{lk}^j(q)$ is the amount of file transferred for request $j \in F$ on link $(l,k) \in E$ in the basic interval q. $b_{lk}(q)$ is the bandwidth available on link (l,k) in the basic interval q. Equation 2 ensures that for each transfer request $j \in F$, for each node l that is neither the source nor the destination node, and for each basic interval q, $0 \leq q \leq i$, the amount of file j that leaves node l equals the amount that enters this node; i.e., nodes other than the source or destination may not create or store data and data cannot be buffered at these nodes for transfer in later basic intervals. Equation 3 requires the source node of request j to send a net f_j units of file j out over all permissible basic intervals and requires the destination node to receive a net f_i units. Equations 4 and 5 ensure that the amount of traffic on each link in each basic interval does not exceed the available capacity of any link in any basic interval. Equation 6 ensures that file transfer amounts are non-negative in permissible basic intervals and Equation 7 ensures that the file transfer amounts are 0 in non-permissible basic intervals.

One may verify that each solution to Equation 2 through 7 defines a valid file transfer schedule for all requests in F and that the finish time of this schedule is at most ft. Further, the inclusion of Equation 1 determines the minimum finish time under the constraint that no file transfer may take place in intervals $q > i$. Also, Equations 2 through 7 have no feasible solution iff the file transfers cannot be scheduled so as to complete by time T_{i+1}.

As noted above, a binary search over the basic intervals is needed to determine the interval where $minFinishTime$ is located and also exact value of $minFinishTime$. This requires to solve $O(\log N)$ LPs, where N is the number of file transfers previous scheduled in the basic intervals $i \geq 0$.

Although the $f_{lk}^j(q)$s that determine $minFinishTime$ define a file transfer schedule that achieves this finish time, these $f_{lk}^j(q)$s may define a transfer schedule that includes cycles. That is, we have portions of a file being moved from node a to node b and back to node a, for example, in the same basic interval. While these cyclic flows do not negatively impact the overall finish time, they affect available bandwidth capacity and so negatively impact our ability to schedule file transfers in future periods.

In Step 2, we overcome the deficiencies of the file transfer schedule obtained from Step 1 by using a slightly different LP formulation in Equations 8 through 13. In this formulation, we minimizes the sum of the $f_{lk}^j(q)$ values across all basic intervals. The value $U = minFinishTime$ computed in Step 1 is used to limit the file transfers' start and end times. We also use i to denote the basic interval for which $T_i \leq minFinishTime \leq T_{i+1}$. It is obvious that the solution to Equation 9 though 13 may contain no cycle, or it can not be

optimal, since we can always remove the cycles and produce a better solution if any cycle exists.

We note that while the LP of Equations 1 through 7 is solved for $O(\log N)$ times, Equations 8 through 13 are solved only once. The above two-step periodic batch scheduling algorithm is referred to as algorithm $Batch$.

$$\min \quad \sum_{j\in J}\sum_{(l,k)\in E}\sum_{q=0}^{i} f_{lk}^{j}(q) \tag{8}$$

$$\text{subject to} \sum_{k:(l,k)\in E} f_{lk}^{j}(q) - \sum_{k:(k,l)\in E} f_{kl}^{j}(q) = 0$$

$$\forall j \in F, \forall l \in V, l \neq s_j, l \neq d_j, 0 \leq q \leq i \tag{9}$$

$$\sum_{q=0}^{i}\Big(\sum_{k:(l,k)\in E} f_{lk}^{j}(q) - \sum_{k:(k,l)\in E} f_{kl}^{j}(q)\Big) =$$

$$\begin{cases} f_j & \text{if } l = s_j \\ -f_j & \text{if } l = d_j \end{cases} \quad \forall j \in F \tag{10}$$

$$\sum_{j\in F} f_{lk}^{j}(q) \leq b_{lk}(T_q) * (T_{q+1} - T_q), \forall (l,k) \in E, q \leq i \tag{11}$$

$$f_{lk}^{j}(q) \geq 0, [T_q, T_{q+1}] \subseteq [S_j, U], \forall (l,k) \in E, \forall j \in F \tag{12}$$

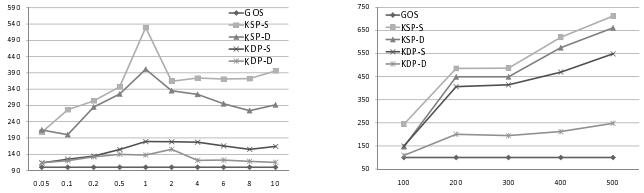
$$f_{lk}^{j}(q) = 0, [T_q, T_{q+1}] \nsubseteq [S_j, U], \forall (l,k) \in E, \forall j \in F \tag{13}$$

VI. Experimental Evaluation

In this section, we measure the performance of the 5 greedy online algorithms of Section IV and the periodic batch algorithm of Section V. For our experiments, we used the 11-node Abilene network [21], the 16-node MCI network [10] and several randomly generated topologies. The bandwidth is 155Mbps for all links in the Abilene network and 100Mbps in the MCI network. The random topologies have 100 to 500 modes and link bandwidths that are randomly selected from the set {50Mbps (OC1), 155Mbps (OC3), 620Mbps (OC12)}. The linear programming problems were solved using the CPLEX package on Intel based workstations. For the KSP and KDP variants of GOS, we set k, the number of paths, to 16. This setting is consistent with the results of [6] and our own results.

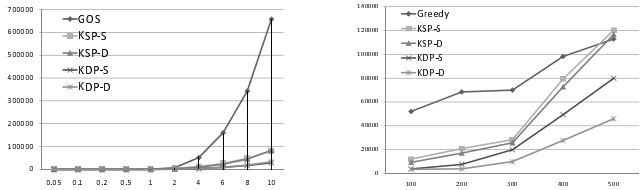
File transfer requests were synthetically generated. Each request is described by the 5-tuple (source node, destination node, file size, request arrive time and start time). The source and destination nodes for each request were selected using a uniform random number generator. The file size is uniformly distributed between 10GB and 100GB. The time at which the request was made followed a Poisson distribution and the arrival rate (request density) varied from 0.05 to 10 requests/time unit. The requested start time was set to be the time at which the request was made plus a random lag. Each of our experiments started with a clean network (i.e., no existing scheduled transfers) and simulated the job arrival process for 1000 time units. So, for example, with a request density of 5 requests/time unit, one run of our experiment would process approximately 5000 requests.

We used the max finish time, i.e. the time when all file transfers in the sequence finish as the performance metric. The performance metric was normalized so that the finish time for the GOS transfers is 100. The run time of an algorithm is measured in milliseconds.



(a) Request Density (b) Network Size

Fig. 3. Maximum finish time for online scheduling.



(a) Request Densities (b) Network Sizes

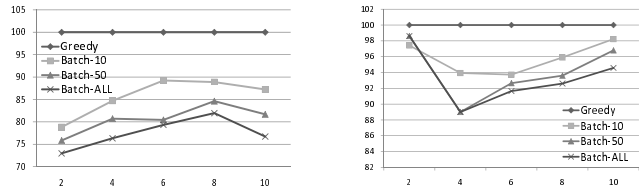
Fig. 4. Run time of online scheduling algorithms

In our experiments, the GOS algorithm results in the smallest maximum finish time consistently and the two KDP algorithms generally outperform the two KSP algorithms, as shown in Figures 3b. Interestingly, we also notice a *peak point* where the relative performance of GOS peaks with respect to the remaining online algorithms. This phenomenon can be explained as follows: when the workload is small, GOS is allowed to provide each job with more bandwidth than KSP and KDP does with little impact on succeeding requests. So, the performance gap increases with request density. However, when more requests arrive within the same time interval, the file transmission processes are more overlapped, As for GOS, providing more for the current job would cause more resource deficit for the late arriving jobs. Hence, when the request density grows larger than the *peak point*, GOS suffers more than other algorithms and cannot outperform the other algorithms as much as before.

When we varied the network size from 100 nodes to 500 nodes, again, GOS consistently yields the smallest maximum finish time (Figure 3a). The advantages of GOS to all the other algorithms keep increasing when we increase network size. This is because GOS generally consumes more bandwidth for each file transfer. As the network size increases, more resources are available for GOS, which accelerates file transfer even more. However, the amount of available bandwidth for other k-path variants is limited by the k-path. Hence, their performance cannot be improved as much as GOS.

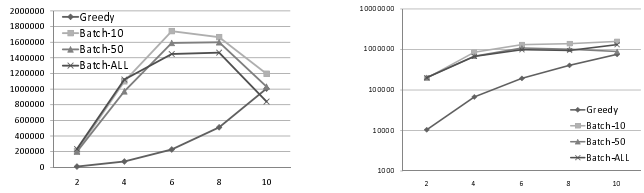
Figure 4 shows the run time of our online algorithms as a function of request density and network size. Although GOS takes more time than other algorithms, its run time, less than one second per request even when request density is 10 req/time unit in a 100-node network, is still acceptable. When the network size grows to 500 nodes, the average process time for each request is still less than 3 seconds in our simulation.

We compared the performance of the periodic batch scheduling algorithm $Batch$ of Section V and our GOS algorithm in two environments: 1) single slice scheduling (SSS) [6], each request has the same value of S_i and in the second environment; 2) multi slice scheduling (MSS), different requests may have



(a) Random, SSS (b) Random, MSS

Fig. 5. $Batch$ and GOS maximum finish times vs. request densities in Abilene and 100-node random network.



(a) SSS (b) MSS

Fig. 6. $Batch$ and GOS run times vs. request densities in 100 nodes random networks

different S_is. In both cases, GOS schedules the requests one by one and in the order in which they arrive while $Batch$ schedules jobs whenever a preset number m of requests has been accumulated or when a preset time interval (in our case 10 units of time) has elapsed.

Figures 5 gives the SSS and MSS maximum finish time for GOS and $m-Batch$ scheduling with m = 10, 50, and 100. The finish time of the $Batch$ schedule has been normalized by the results of GOS. Generally, batch scheduling results in smaller maximum finish times than GOS. In large random networks, batch scheduling results in maximum finish times that are about 10-20% less than those obtained by GOS. As expected, the finish time of batch scheduling reduces as we increase m.

Figures 6 compares the time taken by the $batch$ and GOS algorithms to compute the transfer schedules. When the request density is low, $Batch$ takes 20 times as much time as taken by GOS. This difference decreases as the density increases and when the request density is 10 requests/time unit, $Batch$ takes only about 10-40% more time than GOS does.

VII. Conclusion

We have developed a greedy online scheduling algorithm, GOS, that is optimal in the sense that it minimizes the finish time of the file transfer currently being scheduled. Four variants of this algorithm, $KSP-S$, $KSP-D$, $KDP-S$, and $KDP-D$ have been proposed with the objective of reducing the time required to schedule a file transfer while yielding little in finish time. A two-step periodic batch scheduling algorithm, $Batch$, that employs binary search and linear programming, also has been developed. This algorithm minimizes the maximum finish time of any file transfer in a batch of file transfers. Our experiments show that our GOS algorithm can generate schedules with a maximum finish time slightly larger than those obtained by the periodic batch scheduling algorithm $Batch$. However, GOS generally takes significantly less computation time and its schedules have better mean finish time. Hence, GOS presents a good balance among maximum finish time, mean finish time, and computation time. Further reduction in computation time by sacrificing on maximum finish time and mean finish time may be obtained using one of the proposed four GOS variants. Of these, $KDP-D$ works best. Our current $BATCH$ scheduler minimizes the maximum finish time but does not explicitly consider the mean finish time. In the future, we will develop the $BATCH$ algorithms that also incorporate the mean finish time as the optimization metric.

Acknowledgment

This research is sponsored, in part, by the National Science Foundation under grants ITR-0326155 and XXX.

References

[1] N. S. Rao, S. M. Carter, Q. Wu, W. R. Wing, M. Zhu, A. Mezzacappa, M. Veeraraghavan, and J. M. Blondin, "Networking for large-scale science: Infrastructure, provisioning, transport and application mapping," in *Proceedings of SciDAC Meeting*, 2005.

[2] User Controlled LightPath Provisioning, http://phi.badlab.crc.ca/uclp.

[3] DOE UltraScienceNet: Experimental Ultra-Scale Network Testbed for Large-Scale Science, http://www.csm.ornl.gov/ultranet.

[4] "Internet2," http://www.internet2.edu.

[5] J. Y. Yen, "Finding the k shortest loopless paths in a network," in *Management Science*, 1971.

[6] K. Rajah, S. Ranka, and Y. Xia, "Scheduling bulk file transfers with start and end times," in *6th IEEE International Symposium on Network Computing and Applications*, 2007, pp. 295–298.

[7] E.-S. Jung, Y. Li, S. Ranka, and S. Sahni, "An elvaluation of in-advance bandwidth scheduling algorithms for connection-oriented networks," in *International Symposium on Parallel Architectures, Algorithms, and Networks*, 2008.

[8] U. Black, *MPLS and Label Switching Networks*. Prentice-Hall Pub., 2002.

[9] N. Yamanaka, K. Shiomoto, and E. Oki, *GMPLS Technologies*. CRC Taylor Francis Pub, 2006.

[10] Q. Ma, P. Steenkiste, and H. Zhang, "Routing high-bandwidth traffic in max-min fair share networks," in *ACM SIGCOMM*, 1996, pp. 115–126.

[11] Z. Wang and J. Crowcroft, "Quality-of-service routing for supporting multimedia applications," in *IEEE JSAC*, 1996, pp. 1228–1234.

[12] L. Burchard, "On the performance of networks with advance reservations: Applications, architecture, and performance." in *Journal of Network and Systems Management*, 2005.

[13] R. Guerin and A. Orda, "Networks with advance reservations: The routing perspective," in *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM*, 2000, pp. 118–127.

[14] S. Sahni, N. Rao, S. Ranka, Y. Li, E.-S. Jung, and N. Kamath, "Bandwidth scheduling and path computation algorithms for connection-oriented networks," in *Sixth International Conference on Networking (ICN'07)*, 2007, p. 47.

[15] E.-S. Jung, Y. Li, S. Ranka, and S. Sahni, "Performance evaluation of routing and wavelength assignment algorithms for optical networks," in *13th IEEE Symposium on Computers and Communications*, 2008.

[16] R. Banner and A. Orda, "Multipath routing algorithms for congestion minimization." *IEEE/ACM Trans. Network*, vol. 15, pp. 413–424, 2007.

[17] N. S. V. Rao and S. G. Batsell, "Qos routing via multiple paths using bandwidth reservation." in *INFOCOM*, 1998, pp. 11–18.

[18] Y. Lee, Y. Seok, Y. Choi, and C. Kim, "A constrained multipath traffic engineering scheme for mpls networks," in *Communications, 2002. ICC 2002. IEEE International Conference on*, vol. 4, 2002, pp. 2431 – 2436.

[19] S. Sahni, *Data structures, algorithms, and applications in C++*. Silicon Press, 2005, second Edition.

[20] R. Ahuja, T. Magnanti, and J. Orin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[21] "Abilene," http://abilene.internet2.edu.