Efficient Data Structures for Online QoS-Constrained Data Transfer Scheduling

Mugurel Ionut Andreica, Nicolae Tapus

To cite this version:

Mugurel Ionut Andreica, Nicolae Tapus. Efficient Data Structures for Online QoS-Constrained Data Transfer Scheduling. 7th IEEE International Symposium on Parallel and Distributed Computing (ISPDC), Jul 2008, Krakow, Poland. pp.285-292, 10.1109/ISPDC.2008.36 . hal-00912359

HAL Id: hal-00912359
https://hal.archives-ouvertes.fr/hal-00912359

Submitted on 2 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficient Data Structures for Online QoS-Constrained Data Transfer Scheduling

Mugurel Ionuț Andreica, Nicolae Țăpuș
Politehnica University of Bucharest, Computer Science Department, Bucharest, Romania
{mugurel.andreica, nicolae.tapus}@cs.pub.ro

Abstract

Distributed applications and services requiring the transfer of large amounts of data have been developed and deployed all around the world. The best effort behavior of the Internet cannot offer to these applications the necessary Quality of Service (QoS) guarantees, making the development of data transfer scheduling techniques a necessity. In this paper we propose novel methods of efficiently using some well-known data structures (e.g. the segment tree and the block partition), which can be implemented in a resource manager (e.g. Grid job scheduler, bandwidth broker) in order to serve quickly large numbers of advance resource reservation and allocation requests.

Keywords-data transfer scheduling, block partitioning, segment tree, multidimensional data structures, algorithmic framework, range query, range update.

1. Introduction

In the context of world wide development and deployment of distributed applications and services, QoS guarantees are strictly necessary for achieving good performance levels. In many situations, QoS guarantees are offered by reserving resources in advance. Online resource management systems (e.g. Grid job schedulers, bandwidth brokers) have to deal with many simultaneous reservation requests. In order to provide a good response time, these systems need to use efficient data structures for the management of resource availability and resource reservations. In this paper we present several efficient data structures which can be used for some special classes of resources and many types of reservation requests. The data structures are based on the well-known segment tree and on partitioning the dimensions into equally sized blocks. For some restricted situations, we also present other techniques, based on disjoint sets and balanced trees.

This paper is structured as follows. In Section 2, we present the context of online data transfer scheduling. In Section 3 we present efficient data structure usage for scheduling data transfers on a single link. Section 4 considers the situation of path networks and extends the data structures to this case. In Section 5 we present some practical applications of the data structures we discuss in this paper. Finally, in Section 6 we present related work and in Section 7 we conclude.

2. Online Data Transfer Scheduling

A resource manager receives resource allocation and reservation requests (in our case, data transfer requests) which need to be processed in real time (as soon as they arrive or in batches). A request contains two types of parameters: *constraint parameters* and *optimization parameters*. For each constraint parameter, an acceptable range of values is provided. The optimization parameters need to be either maximized or minimized. Constraint parameters may consist of an earliest start time and a latest finish time, a fixed duration or a minimum (and maximum) amount of resources needed (bandwidth, processors). Optimization parameters may require the minimization of the duration, jitter, congestion or the maximization of the allocated bandwidth.

Many models and algorithms have been developed for the online scheduling problem [1]. In this paper we consider the situation in which the resource manager needs to handle many requests simultaneously and must provide a low response time. Because of the stringent time constraints, the scheduler cannot afford to make elaborate decisions. We consider a simple greedy algorithm which handles the requests in order. For each request, it verifies whether the QoS constraints can be satisfied and, if so, it grants the request; otherwise, the request is rejected. In order to speed up the algorithm, we introduce an algorithmic framework extension for two well-known data structures. We consider a

© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

hierarchical data structure (an extended segment tree) and a non-hierarchical one (based on block partitioning). We will present the algorithmic frameworks for both data structures, as well as several query and update operations that can be supported efficiently. We will mainly be interested in obtaining a good worst case performance for any sequence of queries and updates, but we will also consider the cases of batched updates, followed by online queries, as well as techniques which provide good performance on average. Although the data structures are rather well known, the algorithmic frameworks that we propose are new and solve many problems that were previously handled individually, as well as some new problems that, as far as we are aware, have never been considered before.

3. Scheduling over Time on a Single Link

In this section, we will consider a particular situation, in which the network is composed of only two nodes, connected by a single link. Data transfer requests are sent from both nodes to a central scheduler. The parameters of a request *r* are: the duration of the data transfer D_r, the earliest start time ES_r, the latest finish time LF_r (i.e. if the request is scheduled to begin at time *t*, we must have $ES_r \leq t \leq t+D_r \leq LF_r$) and the minimum required bandwidth B_r. We consider only non-preemptive data transfers (once started, they must not be interrupted). We will consider two *bandwidth models* for the requests. In the first model, each request asks for the whole bandwidth of the link and, thus, we can schedule at most one data transfer at a time. In the second model, each request asks for a fraction of the link's bandwidth. The first model is, obviously, a particular case of the second model, but it can be handled more efficiently in some situations. We will now present several data structures which bring successive performance improvements.

3.1. The Time Slot Array

We will first consider the first bandwidth model (i.e. each data transfer uses the full bandwidth of the link). A common approach is to divide the time horizon into *m* equally-sized time slots and build a time slot array [11] over these slots. For each time slot *t* ($1 \leq t \leq m$), the array *ts* contains an entry *ts[t]*, which can be either *0* (no transfer is scheduled during this time slot) or *1* (a transfer was scheduled during this time slot). Using the time slot division, each transfer is started only at the beginning of a time slot and lasts for an integer number of consecutive time slots (even if the last time slot is not fully occupied, it is still marked as being fully occupied). In order to obtain a good performance, the time horizon must be divided into a large number of time slots (a fine-grained time resolution). In this situation, however, an important aspect to consider is the time it takes to traverse the time slot array for each request. We consider two operations to be performed on the array: a *query* and an *update*. A query verifies if a request can be granted and an update sets all the time slot entries of a data transfer to the same value (*1*, when scheduling a transfer; *0*, when canceling a transfer). The pseudocode of these operations is described below (the time parameters are converted into time slots):

TSAQuery(ES, LF, D):

```
nfree=0
for t=ES to LF do // t=ES, ES+1, …, LF-1, LF
  if (ts[t]=0) then {
    nfree=nfree+1
    if (nfree=D) then return [t-D+1, t] }
  else nfree=0
return "no interval found"
```

TSAUpdate(tstart, D, value):

```
for t=tstart to tstart+D-1 do ts[t]=value // or ts[t]+=value
```

The time complexity of each operations is *O(m)*. The time slot array can be easily enhanced in order to support the second bandwidth model. In this case, it is possible for multiple transfers to be scheduled simultaneously, as long as the maximum bandwidth of the link is not exceeded. The entry *ts[t]* of a time slot represents, in this case, the available bandwidth during that time slot. The update and query functions can be modified in order to run in *O(m)* time. The other data structures will improve the time slot array and maintain the notations and the time slot division model.

3.2. Disjoint Sets for Non-cancelable Reservations

Using the first bandwidth model and if the data transfer reservations cannot be canceled, we can use a disjoint sets data structure [12] in order to maintain the maximal intervals of time slots with an entry equal to *1* in the time slot array. This data structure provides two operations: *Find(t)*, which returns the representative of the disjoint set containing element *t* and *Union(a,b)*, which combines the sets of the elements *a* and *b* into one set (if they are not already in the same set). For each set representative *sr* we maintain two values: *left[sr]* and *right[sr]*, the left and right endpoints of the interval of time slots represented by the set. Within the *Union* procedure we compute the set representatives of the elements *a* and *b*, sr_a and sr_b (sr_a*=Find(a)* and sr_b*=Find(b)*). Then, using some of the well-known

© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

heuristic criteria (like *union by rank* or *union by size*), one of the two representatives (call it sr_u) will be selected as the representative of the combined set. We will set $left[sr_u] = min\{left[sr_a], left[sr_b]\}$ and $right[sr_u]=max\{right[sr_a], right[sr_b]\}$.

DSQuery(ES, LF, D):

```
nfree=0; t=ES
while (t≤LF) do
  if (ts[t]=0) then {
    nfree=nfree+1
    if (nfree=D) then return [t-D+1, t]
    else t=t+1 }
  else { nfree=0; t=right[Find(t)]+1 }
return "no interval found"
```

DSUpdate(tstart, D, value=1):

```
for t=tstart to tstart+D-1 do {
  ts[t]=value
  if ((t>1) and (ts[t-1]=value)) then Union(t-1, t) }
if ((tstart+D≤m) and (ts[tstart+D]=value)) then
  Union(tstart+D-1, tstart+D)
```

The total time complexity of the updates is $O(m\cdot log*(m))$ and the query time is reduced, because we can jump over large intervals of occupied time slots.

3.3. Using Balanced Trees

We will now present a further improvement for the first bandwidth model, based on maintaining a balanced tree *T* of maximal intervals of time slots. A maximal interval is a set of consecutive time slots *t*, for which all the entries *ts[t]* are equal (to *0* or to *1*); if they are equal to *0*, it is a *0-interval* (and its *color* is *0*), otherwise, it is a *1-interval* (and its *color* is *1*). The set of time slots *1,…,m* always has a unique decomposition into maximal intervals. The tree *T* provides three functions: *insertInterval(a, b, c)*, which inserts the interval *[a,b]* having color *c* into *T*, *removeInterval(a, b)*, which removes from *T* the interval *[a,b]*, and *getIntervalContaining(t)*, which returns the interval (and its color) containing the time slot *t*. We present the pseudocode of the query and update operations below. A query finds an unoccupied interval of at least *D* slots.

BTQuery(ES, LF, D):

```
t=ES
while (t≤LF) do {
  (left, right, color)=T.getIntervalContaining(t)
  $left_1$=max{left, ES}; $right_1$=min{right,LF}
  if ((color=0) and ($right_1$-$left_1$+1≥D)) then
    return [$left_1$, $left_1$+D-1]
  else t=right+1 }
return "no interval found"
```

BTUpdate(tstart, D, value):

```
t=tstart
while (t≤tstart+D-1) do {
  (left, right, color)=T.getIntervalContaining(t)
  $left_1$=max{left, tstart}; $right_1$=min{right, tstart+D-1}
  T.removeInterval(left, right)
  if (left<$left_1$) then T.insertInterval(left, $left_1$-1, color)
  if ($right_1$<right) then
    T.insertInterval($right_1$+1, right, color)
  t=right+1 }
T.insertInterval(tstart, tstart+D-1, value)
```

An update fully colors the interval *[tstart, tstart+D-1]* with the color *value*. We cannot offer any guarantees regarding the running time of the query operation. Obviously, it could be $O(m\cdot log(m))$, but, in practical settings, it will run much faster, because it jumps over whole maximal intervals of the same color. The time complexity of the update operation is *O(log(m))*, because the time slot interval *[tstart, tstart+D-1]* intersects only one interval in *T*. However, the implementation of the operation is more general and allows for the interval *[tstart, tstart+D-1]* to intersect multiple other intervals – the maximal interval decomposition is properly maintained in this case, too.

When all the queries have *ES=1* and *LF=m* (i.e. there are no constraints regarding the earliest start time and latest finish time), we can provide stronger guarantees for the running time of the update operation, by maintaining a max-heap (priority queue) with the lengths of the *0-intervals*. Thus, the interval with the maximum length can be obtained in *O(1)* time, because it is located at the top of the heap. Then, by comparing the length of this interval with the required length *D*, we can decide if an appropriate interval of length *D* exists. Using a max-heap slightly complicates the update function. The easiest way to introduce the max-heap is to modify the definitions of the functions *removeInterval* and *insertInterval* of the balanced tree *T*. When a *0-interval* is removed from *T*, we also remove it from the heap; when a *0-interval* is inserted in *T*, we also insert its length in the heap. This way, the time complexity of the query operations becomes *O(1)* and that of the update becomes *O(log(m))*.

3.4. Using the Block Partitioning Method

We can divide the *m* time slots into *m/k* blocks of *k* slots each (the last group may contain fewer time slots). Then, we can use the block partitioning framework introduced in [9]. For the first bandwidth model, an update operation will consists of a range set operation (setting all the time slots in a range to the same value) and a query operation will consists of a range maximum sum segment query. To be more specific, we associate a value v_t to each time slot *t*. This value will always be either *1* or -∞. An update with the parameter *value=0* sets the values of all the time slots in the

© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

interval *[tstart, tstart+D-1]* to *1*. An update with *value=1*, sets the slots in the same interval to -∞. A query consists of the maximum sum segment query contained in the interval *[ES, LF]*. Such a segment will never contain a value v_t=-∞ (unless no value equal to *1* exists in that range). Thus, the returned interval is, in fact, the longest interval of free consecutive time slots and its length can be compared to the parameter *D*. With the framework in [9], we obtain a time complexity of *O(k+m/k)* for each query and update (i.e. *O(sqrt(m))* for *k=sqrt(m)*). For *LF-ES+1=D* and considering the second bandwidth model, we can use the same framework. In this case, the operations used are: range addition update and range minimum query.

3.5. Using an Extended Segment Tree

The segment tree [2] is a binary tree structure used for performing operations on an array *v* with *m* cells. Each cell *i* (*1≤i≤m*) contains a value v_i. Each node *p* of the tree has an associated interval *[p.left, p.right]*, corresponding to an interval of cells. If the node *p* is not a leaf, then it has two sons: the left son (*p.lson*) and the right son (*p.rson*). The interval of the left son is *[p.left, mid]* and the right son's interval is *[mid+1, p.right]*, where *mid=floor((p.left+p.right)/2)*.

The leaves are those nodes whose associated interval contains only one cell. The interval of the root node is *[1,m]*. The height of the segment tree is *O(log(m))*. The tree can be built in *O(m)* time. Query operations consist of computing a function on the values of a range of cells *[a,b]* (range query).

Range Query(a, b): compute $qFunc(v_a, v_{a+1}, ..., v_b)$.

Analogously, we have range updates:

Range Update(u, a, b): v_i*=uFunc(u,* v_i*), a≤i≤b.*

The functions *qFunc* (query function) and *uFunc* (update function) must be at least binary and associative. In order to perform an update we call the function *STrangeUpdate* with the segment tree root as its node argument and the appropriate update parameter. In order to query the segment tree, we call the function *STrangeQuery*, with the segment tree root as its node argument and the left and right cells of the query range. A query/update range is decomposed into *O(log(m))* intervals, corresponding to *O(log(m))* "covering nodes" of the tree (the query/update call stops at these nodes and does not go further down the tree). Besides the covering nodes, all the nodes on the path from the root to each covering node are visited (*O(log(m))* nodes overall). We would like to use the segment tree in order to perform the same operations as in the previous subsection, i.e. range set update and range maximum sum segment query. An easy-to-use segment tree algorithmic framework was introduced in [10]. Each node of the segment tree maintains two values: *uagg* and *qagg*. *uagg* is the aggregate of all the update parameters of the update calls which "stopped" at that node. *qagg* is the query answer for the interval of cells corresponding to the current node, ignoring all the update calls which "stopped" further up in the tree (an update call which "stopped" at one of the current node's ancestors affects the interval of cells of the current node, but its effects are not considered in the *qagg* field of the current node). However, this framework cannot support the kind of range queries and updates we are interested in. The problem lies in the very fact that, for the particular type of queries and updates we are interested in, the updates are not "visible" further down in the tree. Because of this, we augment the segment tree framework with an extra function, called *STpushUpdates*. Basically, the framework is the same, except that update aggregates are pushed down to the two sons (and then cleared) on every update and query call. We would, in fact, like to push these updates all the way towards the leaves, but doing this on every update would take *O(m)* time. Instead, we "piggy-back" future update calls and push the update aggregates "on demand", without affecting the *O(log(m))* complexity of the update/query function calls. We also use a multiplication operator *mop*, which estimates the effects of an update on a range of cells.

```
STpushUpdates(node):
if (node.left<node.right) then {
  STrangeUpdateNodeFit(node.lson, node.uagg)
  STrangeUpdateNodeFit(node.rson, node.uagg)
  node.uagg=uninitialized }

STrangeUpdate(node, u, a, b):
STpushUpdates(node)
if ((a=node.left) and (node.right=b)) then
  STrangeUpdateNodeFit(node, u)
else {
  lson, rson = left and right son of the current tree node
  if ((a≤lson.right) and (lson.left≤b)) then
   STrangeUpdate(lson,u,max(a,lson.left),min(b, lson.right))
  if ((a≤rson.right) and (rson.left≤b)) then
   STrangeUpdate(rson,u,max(a,rson.left),min(b,rson.right))
  STrangeUpdateNodeIncl(node, u, a, b) }

STrangeUpdateNodeFit(node, u):
node.uagg=uFunc(u, node.uagg)
node.qagg=uFunc(mop(u,node.left,node.right),node.qagg)

STrangeUpdateNodeIncl(node, u, a, b):
node.qagg=uFunc(mop(node.uagg, node.left, node.right),
qFunc(node.lson.qagg, node.rson.qagg))

STrangeQuery(node, a, b):
STpushUpdates(node)
if (a=node.left and node.right=b) then
  return STrangeQueryNodeFit(node)
```

© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

```
else {
  q=uninitialized
  if ((a≤node.lson.right) and (node.lson.left≤b)) then
    q=qFunc(q, STrangeQuery(node.lson, max(a, node.lson.left), min(b, node.lson.right))
  if ((a≤node.rson.right) and (node.rson.left≤b)) then
    q=qFunc(q, STrangeQuery(node.rson, max(a,node.rson.left), min(b, node.rson.right))
  return uFunc(STrangeQueryNodeIncl(node, a, b), q) }
```

```
STrangeQueryNodeFit(node):
return node.qagg
```

```
STrangeQueryNodeIncl(node, a, b):
return mop(node.uagg, a, b)
```

With this framework we can support many types of pairs of updates and queries, like all those mentioned in [9] and [10] (e.g. *uFunc(x,y)=qFunc(x,y)=(x+y)* or *uFunc(x,y)=qFunc(x,y)=((x+y) modulo M)*). We will focus now on range set updates. In order to perform a range set update, we need to consider tuples of values. For instance, if we want to perform range sum/minimum/maximum queries, each value of a cell is, in fact a pair *(value, time_stamp)*. Successive update parameters obtain increasing time stamps. The *uFunc* and *qFunc* functions which are used by the framework are defined below:

```
uFunc((w_x, t_x), (w_y, t_y)):
if ($t_x>t_y$) then return ($w_x$, $t_x$) else return ($w_y$, $t_y$)
```

```
qFunc((w_x, t_x), (w_y, t_y)):
res=$w_x+w_y$ // or min{$w_x$,$w_y$} or max{$w_x$,$w_y$}
return (res, max{$t_x$,$t_y$})
```

For the range sum query, the multiplication operator is *mop((w,t),a,b)=((b-a+1)·w,t)*; for the range min/max query, the operator is *mop((w,t),a,b)=(w,t)*. For the maximum sum segment query, each cell's value (and update parameter) is a tuple composed of 5 fields: *(totalsum, maxlsum, maxrsum, maxsum, time_stamp)*. If the value corresponds to an interval of cells *[c,d]*, then the first four fields of tuple are defined as follows:

$$\text{totalsum} = \sum_{p=c}^{d} v_p \qquad \text{maxlsum} = \max_{c\le q\le d} \sum_{p=c}^{q} v_p$$

$$\text{maxrsum} = \max_{c\le q\le d} \sum_{p=q}^{d} v_p \qquad \text{maxsum} = \max_{c\le q\le r\le d} \sum_{p=q}^{r} v_p$$

An update parameter tuple is interpreted as corresponding to a one-cell interval whose value is *totalsum* (thus, the fields *maxlsum*, *maxrsum* and *maxsum* are either equal to *totalsum*, if *totalsum>0*, or to *0*, if *totalsum≤0*). The update and query functions, as well as the multiplication operator, are shown below. All of these (range update, range query) combinations are also supported by the block partitioning framework in [9] (although not explicitly stated in the paper). With the extended segment tree framework, we can support every update and query operation on a range of time slots in *O(log(m))* time.

```
uFunc((ts_x, ml_x, mr_x, m_x, t_x), (ts_y, ml_y, mr_y, m_y, t_y)):
if ($t_x>t_y$) then return ($tsl_x$, $ml_x$, $mr_x$, $m_x$, $t_x$)
else return ($ts_y$, $ml_y$, $mr_y$, $m_y$, $t_y$)
```

```
qFunc((ts_x, ml_x, mr_x, m_x, t_x), (ts_y, ml_y, mr_y, m_y, t_y)):
return ($ts_x+ts_y$ , max{$ml_x$ , $ts_x+ml_y$}, max{$mr_y$ , $ts_y+mr_x$}, max{$m_x$, $m_y$ , $mr_x+ml_y$}, max{$t_x$, $t_y$})
```

```
mop((ts_x, ml_x, mr_x, m_x, t_x), a, b):
return ((b-a+1)·$ts_x$, (b-a+1)·$ml_x$, (b-a+1)·$mr_x$, (b-a+1)·$m_x$, $t_x$)
```

3.6. Batched Range Updates and Queries

In some situations, the updates and the queries occur in stages (they are batched). In this case and for some update functions which are *reversible*, it is possible to obtain a better performance than using the data structures presented previously. An update function is reversible if it is either invertible (each value has an inverse value) or if the effects of an update operation on a single value w_i can be reversed without reconsidering all the other update operations.

We will maintain a separate array *uops* with *m+1* locations. Each location consists of a list of entries, initially empty. For each *rangeUpdate(u, a, b)*, we add a *(Start, u, id)* entry to the list *uops[a]* and a *(Finish, u, id)* entry to the *uops[b+1]* list. *id* is a unique identifier of the update operation. Thus, each update is processed in *O(1)* time. The algorithm is showed below:

```
BatchedUpdates():
for uop in UpdateOperationsSet do {
  uops[$a_{uop}$].add((Start, $u_{uop}$, $id_u$))
  uops[$b_{uop}$+1].add((Finish, $u_{uop}$, $id_{uop}$)) }
ds=empty
for pos=1 to m do
  for (posType, u, id) in uops[pos] do
    if (posType=Start) then ds.add((u, id))
    else ds.remove((u, id))
  $w_{pos}$=ds.computeValue()
```

In the case of the range set update function, we will maintain a max-heap data structure, where the elements are compared according to their time stamp. The *computeValue* function returns the value corresponding to the largest time stamp in the max-heap. The time complexity for processing *n* range updates is *O(n·log(n))*. In the case of range addition, the situation is simpler. The data structure maintains the sum of all the update parameters. When calling *ds.add((u, id))*, the sum is incremented by *u* and when calling *ds.remove((u, id))*, the sum is decremented by *u*. *ds.computeValue()* simply returns the sum. The time complexity for processing *n* range updates is *O(n)*. For other invertible functions (like *xor*, *multiplication* and

© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

others), we obtain the same time complexity (the inverse of *xor* is also *xor* and the inverse of *multiplication* is *division*). After computing all the values in the array *w*, this array can be processed for answering different types of range queries [8].

4. Scheduling over Time on a Path

We now consider the particular situation when the network is a path composed of *n* nodes v_1, v_2, …, v_n (and there is a link between every two consecutive vertices in this order). A data transfer request contains, besides the parameters considered in the previous section, the identifiers of the source and the destination nodes. We could maintain a separate data structure for each network link and, for a data transfer request between two nodes *i* and *j*, we could query (and then update) the data structures of all the *O(n)* links in-between *i* and *j*. In many situations, this approach is good enough. However, we can do better than this, by using multidimensional data structures (for instance, two-dimensional structures, where the first dimension corresponds to the network links and the second dimension corresponds to the time slots).

4.1. The d-dimensional Segment Tree

An extended d-dimensional segment tree ($d \geq 2$) is composed of a segment tree for the first dimension in which every node contains two (d-1) dimensional segment trees $T_{covering}$ and T_{total} (instead of the *uagg* and *qagg*, which are stored only in the 1-dimensional segment trees). Considering that each dimension has *O(m)* elements, then the number of nodes in a d-dimensional segment tree is $O(m^d)$. Each tree node will maintain an attribute *dim*, representing the corresponding dimension (the nodes with *dim=1* contain actual values; the other nodes only contain multidimensional data structures). Using the proposed algorithmic framework, we only need to redefine several functions (and add an extra parameter *dr* to *STrangeUpdate* and *STrangeQuery*). A range query/update consists of *d* intervals (one for each dimension): $dr=[l_1,h_1] \times [l_2,h_2] \times \ldots \times [l_d,h_d]$.

```
STrangeUpdateNodeFit(node, u, dr):
if (node.dim>1) then {
  STrangeUpdate(node.$T_{covering}$, u, $l_{node.dim-1}$, $h_{node.dim-1}$, dr)
  STrangeUpdate(node.$T_{total}$, mop(u, node.left, node,right), $l_{node.dim-1}$, $h_{node.dim-1}$, dr) }
else use the 1D STrangeUpdateNodeFit function

STrangeUpdateNodeIncl(node, u, a, b, dr):
if (node.dim>1) then STrangeUpdate(node.$T_{total}$,
                    mop(u,a,b), $l_{node.dim-1}$, $h_{node.dim-1}$, dr)
else use the 1D STrangeUpdateNodeIncl function

STrangeQueryNodeIncl(node, a, b, dr):
if (node.dim>1) then return mop(STrangeQuery
            (node.$T_{covering}$, $l_{node.dim-1}$, $h_{node.dim-1}$), a, b, dr)
else use the 1D STrangeQueryNodeIncl function

STrangeQueryNodeFit(node, dr):
if (node.dim>1) then
  return STrangeQuery(node.$T_{total}$, $l_{node.dim-1}$, $h_{node.dim-1}$, dr)
else use the 1D STrangeQueryNodeFit function
```

A range update modifies the values of *O(log(m))* intervals (tree nodes) in dimension *d*. For each tree node *p* in dimension *d*, the update function is called on the $p.T_{total}$ (and/or $p.T_{covering}$) *(d-1)*-dimensional extended segment trees. Thus, $O(log^d(m))$ tree nodes are visited. A range query aggregates the values of *O(log(m))* covering nodes in dimension *d*. For each covering node, the query function is called on the $p.T_{total}$ (and/or $p.T_{covering}$) *(d-1)*-dimensional extended segment tree, thus taking $O(log^d(m))$ time overall.

We can easily support any combination of range updates and range queries, as long as the range of every query (or every update) consists of only one cell (point query/update). However, supporting both range queries and range updates with an extended segment tree seems to be possible only for a few types of query and update functions (e.g. range addition updates and range sum queries, with $mop(u,a,b)=(b-a+1)\cdot u$). We were unable to extend the 1D update aggregates pushing technique to multiple dimensions (as this would require pushing and merging multidimensional structures, instead of 0-dimensional structures, i.e. numerical values or tuples with a constant number of fields).

4.2. The d-dimensional Block Partition

The block partitioning technique is extended by splitting each dimension into *m/k k*-sized blocks. In one dimension, the block division requires $(k+1)\cdot m/k$ memory locations. In *d* dimensions, the memory size increases to $((k+1)\cdot m/k)^d$. Extending range queries with point updates (and range updates with point queries) to multiple dimensions is rather easy. We only need to redefine some of the framework's functions. Range queries and updates can be supported when using the block division in time $O(3^d\cdot m^{d/2})$, when choosing $k=m^{1/2}$ (a range query/update has time complexity $O(3\cdot m^{1/2})$ in one dimension). Each d-dimensional partition into blocks consists of an array *dp*, where *dp[i]* is a *(d-1)*-dimensional block partition, corresponding to the i^{th} cell of the d^{th} dimension, and two arrays, *Totalp* and *Covp*, where each entry of the arrays is a *(d-1)*-dimensional block partition, corresponding to a block in the d^{th} dimension. As

© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

before, each block partition has a field *dim*, corresponding to its dimension (*dim=1* corresponds to a normal, one-dimensional block partition). Just like in the case of the multidimensional segment tree, a range query/update consists of d intervals (one for each dimension): $dr=[l_1,h_1] \times [l_2,h_2] \times \ldots \times [l_d,h_d]$. We assume the existence of two functions: *BPrangeUpdate(u, a, b, bpart, dr)* and *BPrangeQuery(a, b, bpart, dr)*, similar to the functions introduced in [9]. *bpart* is a block partition (with dimension *bpart.dim*) and *[a,b]* is the range in the dimension *bpart.dim*.

```
BPrangeUpdateFullBlock(blk, u, bpart, dr):
if (bpart.dim>1) then {
  BPrangeUpdate(u,$l_{bpart.dim-1}$,$h_{bpart.dim-1}$,bpart.Covp[blk], dr)
  BPrangeUpdate(mop(u, bpart.left[blk], bpart.right[blk]),
                $l_{bpart.dim-1}$, $h_{bpart.dim-1}$, bpart.Totalp[blk], dr) }
else { bpart.uagg[blk]=uFunc(u, bpart.uagg[blk])
       bpart.qagg[blk]=uFunc(mop(u, bpart.left[blk],
                bpart.right[blk]), bpart.qagg[blk]) }
```

```
BPrangeUpdatePartialBlock(blk, u, a, b, bpart, dr):
if (bpart.dim>1) then BPrangeUpdate(mop(u, a, b),
                $l_{bpart.dim-1}$, $h_{bpart.dim-1}$, bpart.Totalp[blk], dr)
else bpart.qagg[blk]=uFunc(mop(u, a, b), bpart.qagg[blk])
for i=a to b do
  if (bpart.dim>1) then BPrangeUpdate(u, $l_{bpart.dim-1}$,
                $h_{bpart.dim-1}$, bpart.dp[i], dr)
  else bpart.$v_i$=uFunc(u, bpart.$v_i$)
```

```
BPrangeQueryFullBlock(blk, bpart, dr):
if (bpart.dim>1) then return BPrangeQuery($l_{bpart.dim-1}$,
                $h_{bpart.dim-1}$, bpart.Totalp[blk], dr)
else return bpart.qagg[blk]
```

```
BPrangeQueryPartialBlock(blk, a, b, bpart, dr):
if (bpart.dim>1) then qres=mop(BPrangeQuery($l_{bpart.dim-1}$,
                $h_{bpart.dim-1}$, bpart.Covp[blk], dr), a, b)
else qres=mop(bpart.uagg[blk], a, b)
for i=a to b do
  if (bpart.dim>1) then qres=qFunc(qres, BPrangeQuery
                ($l_{bpart.dim-1}$, $h_{bpart.dim-1}$, bpart.dp[i], dr))
  else qres=qFunc(qres, bpart.$v_i$)
return qres
```

4.3. d-dimensional Batched Range Updates and Queries

Batched range update techniques can also be extended to multiple dimensions. *uops* becomes a d-dimensional array. For each update operation, there will be 2^{d-1} *(Start, u, id)* and 2^{d-1} *(Finish, u, id)* entries. The coordinates where these entries are inserted belong to the set $LH=\{l_1, h_1+1\}x\{l_2,h_2+1\} \times \ldots \times \{l_d,h_d+1\}$. For a cell $(c_1,\ldots,c_d)$ belonging to *LH*, we denote by *cnt* the number of positions *i* where $c_i=l_i$. If *cnt* has the same parity as the number of dimensions *d*, then a *Start* entry is inserted for that position; otherwise, we will insert a *Finish* entry. Afterwards, we traverse the cells in increasing order of the dimension *i* ($i=1,\ldots,d$) and for each *i*, from *1* to *m*. For each cell, we consider all the *Start* and *Finish* entries inserted there and then we compute the cell's value, just like in the 1D case.

5. Practical applications

In this section we discuss the practical applicability of the data structures presented in the previous sections. We will present several scenarios, in which using their capabilities will be beneficial.

In the first scenario, the data transfer scheduler receives requests which ask for a fixed amount of bandwidth *B* during a fixed time interval $[t_1,t_2]$. By dividing the time horizon into equally sized time slots, this problem has two components:

- find the minimum available bandwidth among all the time slots in the interval $[s_1,s_2]$, where s_1 and s_2 are the time slots in which t_1 and t_2 are located.
- decrease the available bandwidth with the same value *B* for each time slot in $[s_1,s_2]$.

These two operations are equivalent to a range addition update and a range minimum query and can be efficiently handled by the proposed data structures.

In the second scenario, clients need to send multimedia data at a minimum rate *R* in a wireless network. The scheduler manages a set of *N* frequencies, numbered consecutively from *1* to *N*. For each frequency *i*, the scheduler keeps track of the maximum rate r_i at which data can be sent on that frequency. We can consider the values $r_i<R$ as negative and the others positive (by setting $r_i=r_i-R$). A request asks for an allocation of consecutive frequency numbers completely located inside an interval $[f_{min}, f_{max}]$, such that the sum of the transfer rates on the frequency interval is maximum. Occasionally, the values r_i may need to be updated. This problem is the range maximum segment sum that we presented.

In the third scenario, the scheduler manages two resources: frequency numbers and time. For each pair *(i,j)*, the available bandwidth $B_{i,j}$ for frequency *i* during time slot *j* is known. Clients need to send as much data as possible during a fixed time slot interval $[s_1,s_2]$ and using a fixed interval of frequency numbers $[f_1,f_2]$. The total amount of data that can be sent is $\sum_{i=s1}^{s2}\sum_{j=f1}^{f2} B_{i,j} \cdot slot\_duration$, where *slot_duration* is the duration of a time slot. A client may want to query several time and frequency intervals until it finds enough available bandwidth. Occasionally, the values

© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

$B_{i,j}$ need to be updated. This problem can be handled efficiently by a multidimensional data structure.

6. Related Work

The segment tree [2] and the partitioning of a range of cells into equally sized blocks have been known for quite some time. Extended variants of these structures have been used in many fields, like computational geometry [6], machine vision, data management in OLAP applications [3,4,5] and advance resource reservations [13]. Usually, they only solve particular problems, for only one type of update and query, or even just for the static case (without any updates) [7]. Moreover, the most commonly studied case corresponds to range queries and point updates. The more general case of range queries and range updates is rarely discussed. We are not aware of other attempts to put together the capabilities of the two data structures into a comprehensive, easy to use algorithmic framework, except [9, 10]. We are also not aware of any extensive analysis of the data structures, from the point of view of the types of problems they can solve.

7. Conclusions and Future Work

In this paper we considered the situation in which a resource manager has to serve quickly and efficiently data transfer requests for a single network link or path. The requests contain hard QoS constraints (which cannot be violated). For this situation, we presented the use of some standard data structures (time slot array, disjoint sets, balanced trees) and we developed new extensions and algorithmic frameworks for two well-known data structures: a hierarchical one (the segment tree) and one which is based on block partitioning, thus continuing the work in [9] and [10]. The data structures are capable of handling several non-trivial variants of range queries and updates. We also extended the data structures to multiple dimensions, but the efficiency decreases with the increase in dimension. We were unable to make the extended segment tree support all the pairs of range queries and updates that can be supported in one dimension. The use of the framework was presented by concrete examples, like the range maximum sum segment query and the range set update, which were not considered anywhere else in this form (as far as we are aware). The batched update case for one dimension and range sum addition appears to be part of the algorithmic folklore, but we are not aware of its extensions to other update operations. In the end, we also showed how the data structures can be used in the process of resource allocation and reservation, by presenting motivating scenarios.

8. References

[1] K. Pruhs, J. Sgall, and E. Torng, *Online Scheduling*, CRC Press, 2004.

[2] K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[3] S.-J. Chun, C.-W. Chung, J.-H. Lee and S.-L. Lee, "Dynamic Update Cube for Range-Sum Queries," *Proceedings of the 27th International Conference on Very Large Data Bases,* pp. 521-530, 2001.

[4] C. K. Poon, "Dynamic Orthogonal Range Queries in OLAP," *Theoretical Computer Science*, vol. 296 (3), 2003.

[5] H.-G. Li, T. W. Ling, S. Y. Lee, and Z. X. Loh, "Range Sum Queries in Dynamic OLAP Data Cubes," *Proc. of the 3rd Intl. Symposium on Cooperative Database Systems for Advanced Applications (CODAS)*, pp. 74-81, 2001.

[6] J. L. Bentley, "Multidimensional divide-and-conquer," *Communications of the ACM*, vol. 23 (4), pp. 214-229, 1980.

[7] K.-Y. Chen, and K.M. Chao, "On the Range Maximum-Sum Segment Query Problem," *Discrete Applied Mathematics,* vol. 155 (16), pp. 2043-2052, 2007.

[8] M. A. Bender, and M. Farach-Colton, "The LCA Problem revisited," *Proc. of the 4th Latin American Symposium on Theoretical Informatics, LNCS*, vol. 1776, pp. 88-94, 2000.

[9] M. I. Andreica, "Optimal Scheduling of File Transfers with Divisible Sizes on Multiple Disjoint Paths," *Proc. of the IEEE Romania Intl. Conf. "Communications"*, pp. 155-158, 2008.

[10] M. I. Andreica, and N. Tapus, "Optimal TCP Sender Buffer Management Strategy," *Proc. of the IEEE/IARIA Intl. Conf. on Communication Theory, Reliability and Quality of Service,* pp. 41-46, 2008.

[11] L.-O. Burchard, "Analysis of Data Structures for Admission Control of Advance Reservation Requests", *IEEE Transactions on Knowledge and Data Engineering*, vol. 17 (3), pp. 413-424, IEEE Press, 2005.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press and McGraw-Hill, 2001.

[13] A. Brodnik, A. Nilsson, "An Efficient Data Structure for Advance Bandwidth Reservations on the Internet", *Proc. of the 3rd Conference on Comp. Sci. and Electrical Eng.*, 2002.

© 20xx IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.