BHEFT - An analysis through Implementation

Big data is everywhere. It needs an efficient way to be sorted through and processed in both a timely and uncostly fashion. This is known as the scheduling problem, deciding the best way to execute a node that exists in a sea of hundreds in a way that will mesh well with others. This can be done with BDC, budget-deadline constraint, planning. A bicriteria strategy that finds the lowest cost for the lowest possible execution time in a workflow. BHEFT, Budget-constrained Heterogeneous Earliest Finish Time, is an algorithm that fits this bicriteria. In this report, an implementation of BHEFT is discussed as well as the specific details of BHEFT that make it unique when compared to other well known scheduling algorithms. It is found that BHEFT is an effective scheduling algorithm compared to other well known ones especially when under heavy load. GCC (Gator Computational Cloud), a web based cloud workflow executor, is also briefly discussed as the testing grounds for the aforementioned BHEFT algorithm

INTRODUCTION

In the recent years of modern computing, cloud based applications have become an integral part of delivering a complete user experience to customers all over the world. As one might guess, the constant traffic among users on the same network can cause lots of problems, especially when they're all trying to access the same resource. This same idea can be applied to the execution of various data analysis algorithms on big data, thousands of calculations being done on just one machine. Spreading out this workload between different stages and machines is optimal for a case such as this. This is called the scheduling problem; finding out a way to efficiently execute all the necessary computations on data in a way that is both optimized for cost and time efficiency.

.In any industry when you have the ability to optimize, many will flock to solve this problem especially when money is on the line. Take for example a financial institution that has hundreds of thousands of transactions coming from their customers. If the processing of these lags behind at any point they could lose a customer that is fed up at the denial of their credit card. Many solutions to the scheduling problem have been proposed but none of them are decidedly optimal. All the recent solutions have their upsides and downsides.

Among these, there is the Heterogeneous Earliest Finish Time (HEFT) algorithm, a very well known scheduling heuristic which aims at minimizing the execution time at any cost. Of course this may be taken as optimal for those that don't care about cost, but this may change from customer to customer. Sometimes it can be a tradeoff: a customer could have a deadline which is necessary to meet in order to complete a monetary transaction. In the other case that time and money may be considered to be at the same priority, also known as a budget-deadline constraint, the Budget-constrained Hetero-geneous Earliest Finish Time (BHEFT) algorithm may be considered. The focus of the BHEFT algorithm is exactly as it seems, finish at the earliest possible time for a given budget.

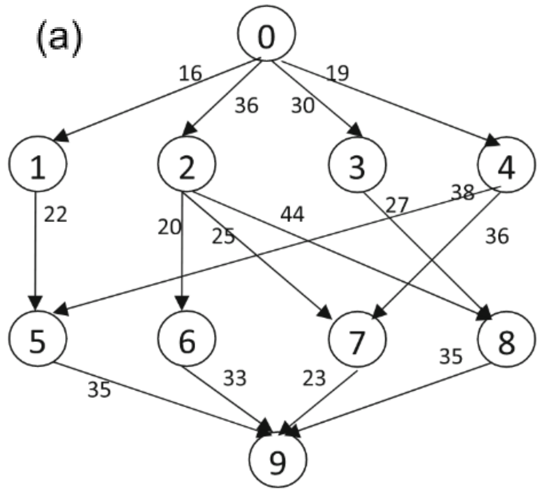
The benefits of an algorithm such as BHEFT can not be minimized. The largest benefit is accessibility to a variety of users by constraining its use by a given budget. This way, a user has the ability to determine whether or not a service may be right for them based on their budget. Not on that, but BHEFT gives the user the ability to set any budget that may work for them. This way, the customers that only have time as their highest priority and are satisfied with any budget may set a stupendously large one while still ending up with the same results that the previously mentioned HEFT algorithm which just focused on earliest finish time.

In this paper a general description of BHEFT will be discussed as well as its implementation into the web-based workflow execution platform called GCC (Gator Computational Cloud). After this description, preliminary results will be shown showing the efficacy of BHEFT's performance in GCC followed by a related work section, focusing on the advent of the BHEFT algorithm in the research paper which was used as a guide for implementation [1]. Finally, there will be a conclusion summarizing the discovered results and any directions for future work brought about by them.

BHEFT IMPLEMENTATION

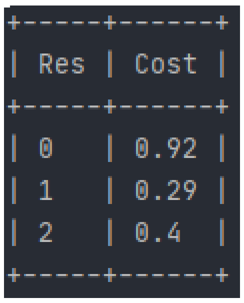
Both HEFT and BHEFT are algorithms executed upon DAG (Directed Acyclic Graphs), a graph which has a specific beginning and end point. No matter which point you choose, the ending state will always be the same. This means that there are two key concepts to understand, nodes and edges between nodes.

In figure (a), node 0 is the beginning and node 9 is the end. Every single node is connected to at least one other with an edge that contains a weight. The weight in the case of BHEFT is the amount of data being transferred between nodes. The data is created by each node from executing analysis on data from its parent(s). This means the initial data for node 0 is supplied by the user and the results from node 9 are transferred back to the user as the final results.



The scheduling problem is created when there are different services that may be used to execute the data analysis on each node. Some services may be better at particular types of data manipulation than others, creating an obvious "best service" for each one which produces the quickest possible finish time (b). BHEFT's job is to determine the nodes which "best service" saves the most money while taking into account time spent. This idea is more clearly demonstrated in figure (c) which shows each services' cost per second. In order to find

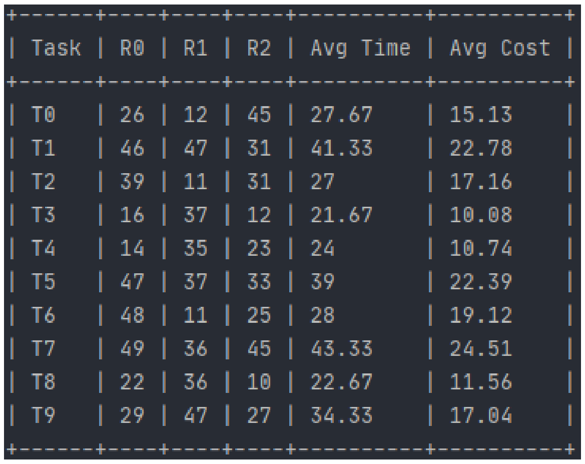
(b)



```
+-----+------+
| Res | Cost |
+-----+------+
| 0   | 0.92 |
| 1   | 0.29 |
| 2   | 0.4  |
+-----+------+
```

the best service there first must be a test done on each node with every single service. The R0, R1, and R2 columns represent the time taken for each task to be completed on their respective service. This data can be taken from a small sample size, estimated by the user, etc.

(c)



```
+------+----+----+----+----------+----------+
| Task | R0 | R1 | R2 | Avg Time | Avg Cost |
+------+----+----+----+----------+----------+
| T0   | 26 | 12 | 45 | 27.67    | 15.13    |
| T1   | 46 | 47 | 31 | 41.33    | 22.78    |
| T2   | 39 | 11 | 31 | 27       | 17.16    |
| T3   | 16 | 37 | 12 | 21.67    | 10.08    |
| T4   | 14 | 35 | 23 | 24       | 10.74    |
| T5   | 47 | 37 | 33 | 39       | 22.39    |
| T6   | 48 | 11 | 25 | 28       | 19.12    |
| T7   | 49 | 36 | 45 | 43.33    | 24.51    |
| T8   | 22 | 36 | 10 | 22.67    | 11.56    |
| T9   | 29 | 47 | 27 | 34.33    | 17.04    |
+------+----+----+----+----------+----------+
```

The average time and cost are then calculated to aid in the next, more complicated equations further below. The pseudocode below (d), from the aforementioned BHEFT paper by Wei Zheng and Rizos Sakellariou [1], will be accessed line by line, introducing the necessary equations when applicable.

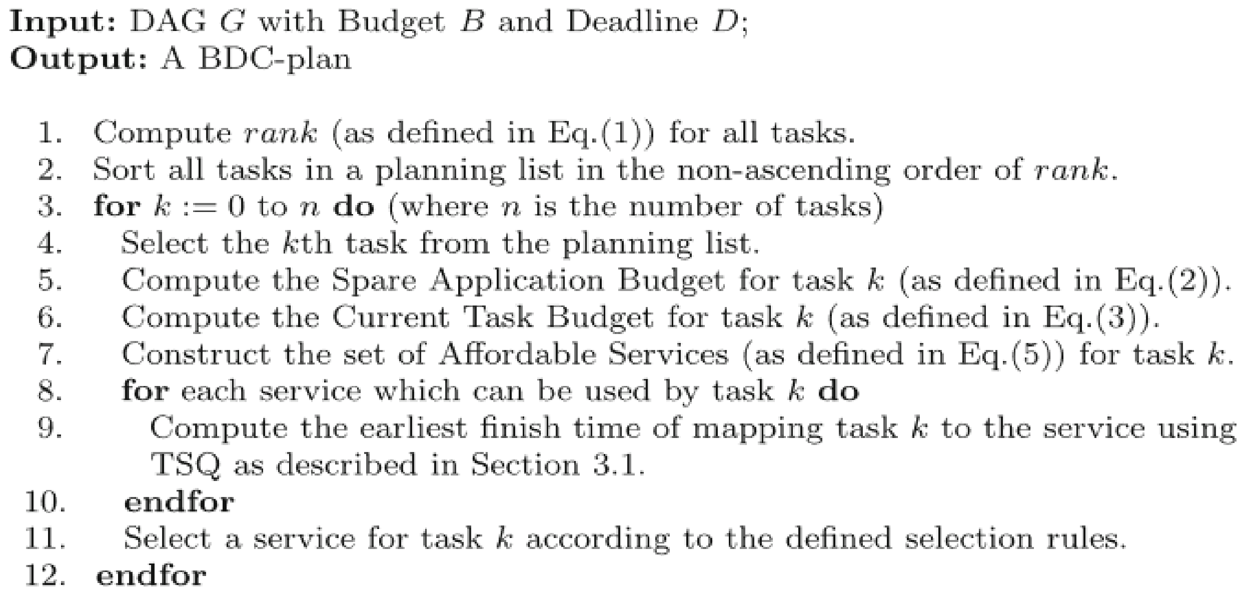
Line 1 introduces *rank*, a value calculated using upward ranking which adds the rank from child nodes recursively all the way to the bottom. The purpose of finding the max child (Eq. 1) for each node is to show that if a node has more than one parent, it must wait until **all** of its parents are done executing. Because of this, another word for parent nodes is **dependencies**. The rank of a task is defined by

$$rank_i = \overline{et}_i + \max_{j \in Succ(i)} \left\{ \overline{dt}_{i,j} + rank_j \right\} \qquad (1)$$

where Succ(i)is the set of the child tasks of task i, $\overline{et_i}$ is the average execution time of t_i, $\overline{dt_{i,j}}$ is the average data transfer time of edge $t_i \rightarrow t_j$. In the case of childless nodes, the rank equals the average execution time.

After the rank is calculated the nodes are sorted according to their rank. This is now the order that the tasks will be used for these next calculations. In line 3 the tasks are iterated upon with lines 4 to 11. Within these lines there are three key components necessary for BHEFT: Spare Application Budget (SAB), Current Task Budget (CTB), and finally Adjustment Factor (AF). It's

(d)



```
Input: DAG $G$ with Budget $B$ and Deadline $D$;
Output: A BDC-plan

 1. Compute $rank$ (as defined in Eq.(1)) for all tasks.
 2. Sort all tasks in a planning list in the non-ascending order of $rank$.
 3. for $k := 0$ to $n$ do (where $n$ is the number of tasks)
 4.   Select the $k$th task from the planning list.
 5.   Compute the Spare Application Budget for task $k$ (as defined in Eq.(2)).
 6.   Compute the Current Task Budget for task $k$ (as defined in Eq.(3)).
 7.   Construct the set of Affordable Services (as defined in Eq.(5)) for task $k$.
 8.   for each service which can be used by task $k$ do
 9.     Compute the earliest finish time of mapping task $k$ to the service using
        TSQ as described in Section 3.1.
10.   endfor
11.   Select a service for task $k$ according to the defined selection rules.
12. endfor
```

important to understand that the CTB_k is the determining factor in which services are added to S_k* which is a set of services that cost less than or equal to the CTB.

To begin the SAB on line 5 is defined as

$$SAB_k = B - \sum_{i=0}^{k-1} c_i - \sum_{j=k}^{n-1} \bar{c}_j \tag{2}$$

where B is the user provided budget, $\overline{c_i}$ is the reservation cost of the allocated task i, $\overline{c_j}$ is the average reservation cost of the unallocated task j over different resource mappings, and n is the number of tasks. The SAB_k symbolizes the expected amount of extra money that can be spent on that particular task. The first sigma is the sum of the actual cost of the tasks scheduled thus far and the second one is the sum of the average costs for the future tasks. That just leaves the present node, which, if added, would equal the budget. Next, the AF_k, it should be between line 5 and 6, is defined as

$$AF_k = \begin{cases} \bar{c}_k / \sum_{i=k}^{n-1} \bar{c}_i & : \quad SAB_k \geq 0 \\ 0 & : \quad SAB_k < 0 \end{cases} \tag{4}$$

where $\overline{c_k}$ is the average cost of task k and $\overline{c_i}$ is the sum of the average unallocated future tasks including k, but this is only the case if the SAB_k is greater than or equal to 0. The AF_k is a value that determines what percentage of the SAB_k should be spent on this task. Given this, it should make sense that the final tasks AF_k will always be 1 (If there is spare budget, spend it!). That brings us to our last equation, the CTB on line 6 which is defined as

$$CTB_k = \bar{c}_k + SAB_k \times AF_k \tag{3}$$

where $\overline{c_k}$ is the average cost of task k and both CTB_k and AF_k are what you would expect. Above it was said that S_k* on line 7 is the set of affordable tasks, which are defined as

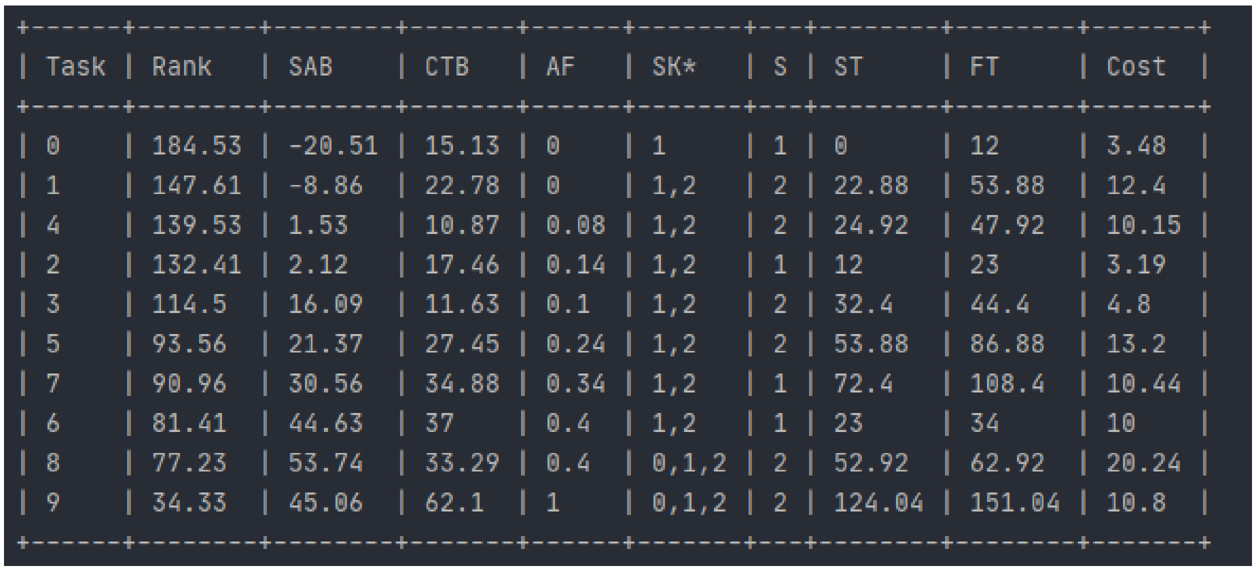
$$S_k^* = \{s_{x,p} | \exists s_{x,p}, c_{k,p} \leq CTB_k\} \tag{5}$$

which means if the time spent on that service (c) multiplied by the respective service cost per second (b) and check to see if it's less than the CTB_k, it belongs to S_k*. Now the "best possible" service can be found using these three rules:

1. If S_k* = ∅, the affordable service with the earliest finish time is selected;
2. If S_k* = ∅, and $SBA_k \geq 0$, the service with the earliest finish time is selected;
3. If S_k* = ∅, and $SBA_k < 0$, the service that is cheapest is selected;

With S selected, see figure (e), BHEFT can now begin determining the ST and FT, start

(e)



```
+------+--------+--------+-------+------+-------+---+--------+--------+-------+
| Task | Rank   | SAB    | CTB   | AF   | SK*   | S | ST     | FT     | Cost  |
+------+--------+--------+-------+------+-------+---+--------+--------+-------+
| 0    | 184.53 | -20.51 | 15.13 | 0    | 1     | 1 | 0      | 12     | 3.48  |
| 1    | 147.61 | -8.86  | 22.78 | 0    | 1,2   | 2 | 22.88  | 53.88  | 12.4  |
| 4    | 139.53 | 1.53   | 10.87 | 0.08 | 1,2   | 2 | 24.92  | 47.92  | 10.15 |
| 2    | 132.41 | 2.12   | 17.46 | 0.14 | 1,2   | 1 | 12     | 23     | 3.19  |
| 3    | 114.5  | 16.09  | 11.63 | 0.1  | 1,2   | 2 | 32.4   | 44.4   | 4.8   |
| 5    | 93.56  | 21.37  | 27.45 | 0.24 | 1,2   | 2 | 53.88  | 86.88  | 13.2  |
| 7    | 90.96  | 30.56  | 34.88 | 0.34 | 1,2   | 1 | 72.4   | 108.4  | 10.44 |
| 6    | 81.41  | 44.63  | 37    | 0.4  | 1,2   | 1 | 23     | 34     | 10    |
| 8    | 77.23  | 53.74  | 33.29 | 0.4  | 0,1,2 | 2 | 52.92  | 62.92  | 20.24 |
| 9    | 34.33  | 45.06  | 62.1  | 1    | 0,1,2 | 2 | 124.04 | 151.04 | 10.8  |
+------+--------+--------+-------+------+-------+---+--------+--------+-------+
```

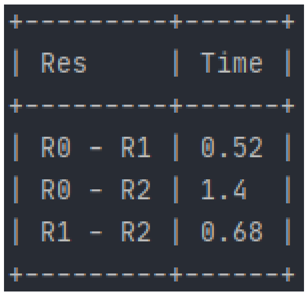
time and finish time respectively, of each task its node. ST is defined as

$$ST_k = max(j \in Sup(k))\{ FT_j + TW_{j,k}\}$$
(6)

$$TW_{j,k} = if(S_k \neq S_j)\{T_{j,k} \times wt_{j,k}\} \qquad (7)$$

Where $max(j \in Sup(k))$is the max ST_k that is calculated of the superiors, FT_j is FT of task j, S_k is service of task k, same for j respectively, $T_{j,k}$is the transfer cost, figure (f) between S_k and S_j , and $wt_{j,k}$ is the edge weight, figure (a) between nodes j and k. It makes sense that there is only a transfer time between nodes with different services because if they are the same service there would be no need for a transfer.

(f)



```
+---------+------+
| Res     | Time |
+---------+------+
| R0 - R1 | 0.52 |
| R0 - R2 | 1.4  |
| R1 - R2 | 0.68 |
+---------+------+
```

IMPLEMENTATION

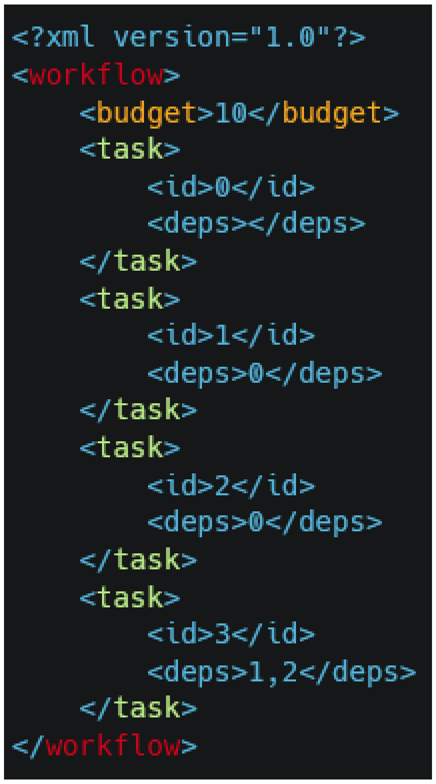
The implementation of BHEFT was coded using java with a focus on the principle of object oriented programming by using objects to model the BHEFT scheduling algorithm in an intuitive way. The primary examples of this are the BNode object, representing a node in the DAG (a), and the res object which was used to symbolize a resource (or service). The following explanation of the implementation will resolve in the order of how the program would run after it was implemented into the web application of GCC.

Right before the necessary information is passed from the front to the backend, the file ExecDriver.java is run which contains the main method that executes the whole program. The arguments which are provided are as follows: <user> <workflow file name> <workflow results file name> and 3 others that contain credentials for AWS.

The first argument, <user>, is a username that the user chooses when creating an account on the GCC website. All of their data pertaining to BHEFT workflow execution is stored here, some of it being workflows file, logs from past executions, and more.

The second argument, <workflow file name> is a particularly structured xml file which GCC and BHEFT use to store necessary information pertaining to the DAG such as dependencies for each node as well as the budget constraint specified by the user.

(g)



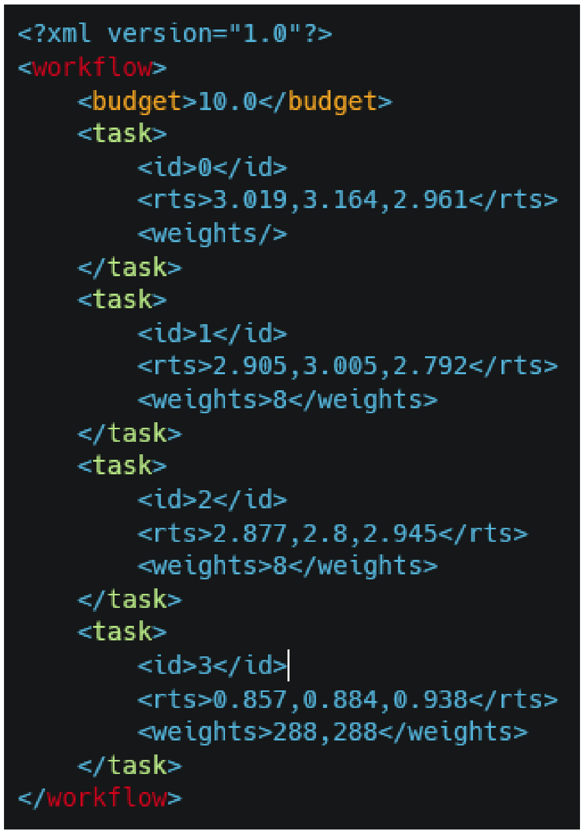
```
<?xml version="1.0"?>
<workflow>
    <budget>10</budget>
    <task>
        <id>0</id>
        <deps></deps>
    </task>
    <task>
        <id>1</id>
        <deps>0</deps>
    </task>
    <task>
        <id>2</id>
        <deps>0</deps>
    </task>
    <task>
        <id>3</id>
        <deps>1,2</deps>
    </task>
</workflow>
```

Figure (g) is an example of a very simple workflow, containing 4 different tasks that create a diamond pattern when their dependencies are considered. This file is read in and converted to aforementioned objects, preparing for execution.

The third argument, <workflow results file name>, is an xml file which contains tasks just like arg 2 but it contains data specifically needed for BHEFT to run.

This is the file titled <input workflow name>BHEFT.xml generated by the backend when "test" is specified for this third argument. When doing so, the backend will execute the code on each node for each service which is currently available. In reality, this would be a hugely ineffective means to save money and these results

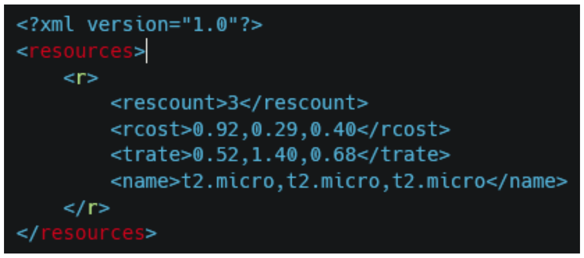
(h)



```
<?xml version="1.0"?>
<workflow>
    <budget>10.0</budget>
    <task>
        <id>0</id>
        <rts>3.019,3.164,2.961</rts>
        <weights/>
    </task>
    <task>
        <id>1</id>
        <rts>2.905,3.005,2.792</rts>
        <weights>8</weights>
    </task>
    <task>
        <id>2</id>
        <rts>2.877,2.8,2.945</rts>
        <weights>8</weights>
    </task>
    <task>
        <id>3</id>
        <rts>0.857,0.884,0.938</rts>
        <weights>288,288</weights>
    </task>
</workflow>
```

would probably be generated some other way, either through estimation or running the "test" execution on data a fraction of the size so as to not waste too much money. The variable rts is a list of the durations for the test of each service on a node in order

(i)



```
<?xml version="1.0"?>
<resources>
    <r>
        <rescount>3</rescount>
        <rcost>0.92,0.29,0.40</rcost>
        <trate>0.52,1.40,0.68</trate>
        <name>t2.micro,t2.micro,t2.micro</name>
    </r>
</resources>
```

according to the next xml necessary for execution.

The resource file defines all of the services with their respective costs per unit of data, transfer rate between each other, and name. rcosts is defined in ascending order, assigning id's starting at 0. The next list, trate, has the rates listed in neighbors of 0 in ascending order first. In this case, the order is: 0-1, 0-2, 1-2. If there were 4 services, the order would be: 0-1, 0-2, 0-3, 1-2, 1-3, and 2-3. The list of names is necessary for choosing the correct instance type in aws (in this case, service). Finally, weights are listed in the same order as dependents in the original workflow folder.

In order to make use of these xml files, an xml parser was utilized to convert them into a list of BNodes and resources in java. Everything needed to execute BHEFT upon this workflow is set up correctly.

From here, the process is very similar to the figure (d), the rank is rank is found with the exact same formula, except there is one very important detail, in order to obtain $\overline{dt_{i,j}}$ in equation (1), the transfer time along the edge, you must multiply the edge weight by 0.578. An explicit reason for this was not given in the research paper, but since we know $transferTime = edgeWeight * 0.578$, if we solve for 0.578 the units on the left side of the equation become time/data units. Taking the reciprocal of those units gives data units/time, a more familiar unit. We also have to take the reciprocal of 0.578 which is 1.730, so this is most likely the data transfer speed that Zheng and Sakellariou used.

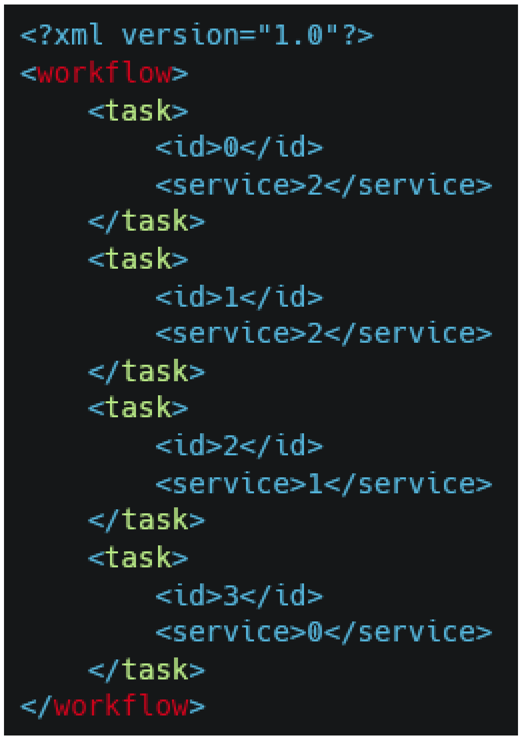
The tasks are then sorted according to rank and calcBHEFT, the primary method of BNode.java which facilitates all other

BHEFT calculations, is called on each node. The code written in this function and its dependents is straightforward except for the findBestService method. It contains 3 main code blocks, each of which corresponds to one of the cases in the rules below equation (5). They iterate over each of the possible services in S_k* with their own conditionals according to the rules. At the end, the elapsed time for the chosen service is recorded in addition to the best possible service.

At this point, everything except for the ST and FT of each task has been calculated. This is the first point that recursion is used in this implementation due to the unique quality of BHEFT's upward ranking system, completely reversing how one typically thinks of recursing through a network of nodes. This ST and FT calculation is the first time that this is not the case, thankfully.

The function in mention is findSTandFT which takes a previous node, a current node, and the FT of the previous node as parameters. While recursing, a check very similar to equation (7) occurs.

(j)



```
<?xml version="1.0"?>
<workflow>
    <task>
        <id>0</id>
        <service>2</service>
    </task>
    <task>
        <id>1</id>
        <service>2</service>
    </task>
    <task>
        <id>2</id>
        <service>1</service>
    </task>
    <task>
        <id>3</id>
        <service>0</service>
    </task>
</workflow>
```

An approach very similar to multithreading is taken here as well to check if the resulting FT is larger than a FT if one already exists. This is done because at the very end of the function a loop on the current node's children is done, calling findSTandFT for each of its children. Even though there is only ever one thread, one can think of it as if multiple threads (children in this case) are trying to update the FT at the same time.

As soon as BHEFT has finished, the algorithm spits out an xml file (j) very similar to figure (g). A service variable is added which is also the best service calculated by BHEFT. When ExecDriver.java is called with the name of this file instead of "test", the workflow will execute once on each node with the specified service.

For ease of testing, a command line table was found online [2] and edited slightly so that it would be able to display every item needed with the correct formatting. These tables are figures (b,c,e, and f).

RESULTS

Testing this implementation of BHEFT opened up many eyes, especially those of myself. After much time and talking with Professor Mohan of Allegheny college, I discovered a core flaw that invalidated the final tests. I was under the impression that the "real" cost of each node was given by the user for some reason. It took a me a very long time to realize this, as I had worked with the BHEFT I thought was correct for so long. The methodology that I used for testing was a google docs spreadsheet to calculate the value of each variable at every single point. The benefit of using a spreadsheet application was that you can do all the calculations right in the program without needing to refer to an external device. I came up with the DAG seen in figure (g). The first two examples that I created are pretty similar to the ones used as examples for previous parts of this technical report so I have decided to share the third example.

The third example is unique because it shows what it looks like when the budget is set too low. The SAB and AF are dead giveaways, especially when you can see that the SAB is negative at the very bottom. This not only represents the lack of spare budget to use on this task but also a negative value means you're forced to spend more than you have. Normally this may be fine, such as in figure (e) where you notice that just the first two are negative, but since these negative SABs are at the front there is time to get out of the negatives. In figure (k) this is not the case.

Another great way to determine if a budget is insufficient is to look at the last tasks AF. If it's not equal to one, then a larger budget is needed.

Now take a look at this corrected BHEFT table (m). When you actually calculate values in their intended way, the results start making sense. Reducing the budget to about 87 the sum of the costs, which is $83.81 yields this next table (n). The sudden jump in Adjustment factor is because there is extra money in the budget, even if it is just 18 cents, that BHEFT chooses to spend to optimize for the deadline constraint since the budget was already met.

Overall, I am disappointed that there was not more testing similar to this. Perhaps if more of this testing was done, I would have caught the issue earlier which meant more testing could have been done.

On the bright side, the end product does work correctly. This can be seen in the xml diagrams above, namely (g and h). These files were created by BHEFT after data was gathered from the 3 tests on each node, one for each service.

In order to get these results bash commands were used to grab text directly from the command line. Bash has a built-in time command that times the execution time of a program so all that was left to do was clean up the output and save it to a file that would be later transferred back to the ExecDriver.java.

(k)

EXAMPLE 3

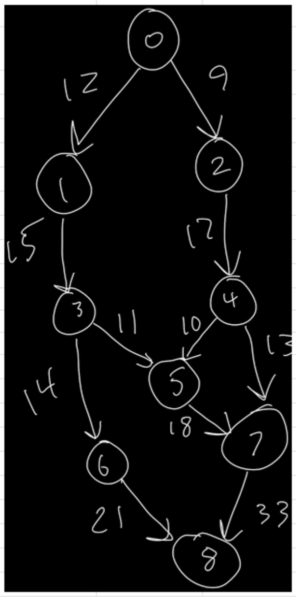
Task	R1	R2	R3	Avg Time	Avg Cost
T0	18	34	20	24.00	15.57
T1	12	35	12	19.67	12.49
T2	32	20	30	27.33	18.70
T3	34	35	24	31.00	20.87
T4	11	6	12	9.67	6.61
T5	12	22	9	14.33	9.34
T7	12	24	12	16.00	10.36
T6	24	12	5	13.67	9.77
T8	22	36	21	26.33	17.24

Res	Time
0 - 1	0.85
1 - 2	0.95
0 - 2	0.73

Res	Cost
0	0.80
1	0.58
2	0.63

Budget	125.00

Task	Rank	SAB	CTB	AF	SK*	S	ST	FT	Cost
T0	182.78	4.04	16.09	0.129	0, 2	0	0.00	18.00	25
T1	151.84	-5.38	12.49	0.000	0, 2	0	18.00	30.00	41
T2	138.75	-33.90	18.70	0.000	1	1	25.65	45.65	23
T3	123.50	-38.20	20.87	0.000	1, 2	2	40.95	64.95	38
T4	101.59	-55.32	6.61	0.000	1	1	45.65	51.65	21
T5	86.14	-69.71	9.34	0.000	2	2	64.95	73.95	53
T7	61.41	-103.01	10.36	0.000	0, 2	0	87.09	99.09	156
T6	52.14	-269.37	9.77	0.000	0, 1	2	64.95	69.95	12
T8	26.33	-261.24	17.24	0.000	2	2	123.18	144.18	10



(m)

Task	Rank	SAB	CTB	AF	SK*	S	ST	FT	Cost
T0	182.78	4.04	16.09	0.129	0, 2	0	0.00	18.00	14
T1	151.84	5.22	12.49	0.201	0, 2	0	18.00	30.00	10
T2	138.75	8.10	20.39	0.209	1,2	1	25.65	45.65	12
T3	123.50	15.20	20.87	0.118	1,2	2	40.95	64.95	15
T4	101.59	20.96	13.86	0.346	0,1,2	1	45.65	51.65	3
T5	86.14	24.09	16.12	0.281	0,1,2	2	64.95	73.95	6
T7	61.41	38.12	20.93	0.277	0,1,2	2	73.95	85.95	8
T6	52.14	20.20	12.28	0.124	1,2	2	64.95	69.95	3
T8	26.33	37.18	54.42	1.000	2	2	85.95	106.95	13

Task	Rank	SAB	CTB	AF	SK*	S	ST	FT	Cost
T0	182.78	-32.96	15.57	0.000	0, 2	0	0.00	18.00	14
T1	151.84	-31.78	12.49	0.000	0, 2	0	18.00	30.00	10
T2	138.75	-28.90	18.70	0.000	1,2	1	25.65	45.65	12
T3	123.50	-21.80	20.87	0.000	1,2	2	40.95	64.95	15
T4	101.59	-16.04	6.61	0.000	0,1,2	1	45.65	51.65	3
T5	86.14	-12.91	9.34	0.000	0,1,2	2	64.95	73.95	6
T7	61.41	1.12	10.67	0.277	0,1,2	2	73.95	85.95	8
T6	52.14	-16.80	9.77	0.000	1,2	2	64.95	69.95	3
T8	26.33	0.18	17.42	1.000	2	2	85.95	106.95	13

(n)

RELATED WORK

As mentioned before, this BHEFT implementation was modeled off of this [1] paper by Wei Zheng and Rizos Sakellarion. All of the equations that were used in this paper are exactly the same as their paper except for the calculations for ST and FT. I found out very recently that their definitions for ST and FT are based off of a cloud platform that only has access to one of each service, so there could not be any parallel processes occurring on the same service.

For my implementation of BHEFT which uses rented cloud servers from aws, it would not make sense to implement it with the same limitation in mind since ec2 instances can be a variety of types at the same time, running as many in parallel as possible.

Their paper goes into far more detail and depth than I have in this one, including results from testing BHEFT head to head with several other well known scheduling algorithms like DCA (dynamic constraint algorithm) a scheduling algorithm which is bicriteria like BHEFT but is based upon a sliding constant to change the weight of constraints [6], LOSS, an algorithm that splits the scheduling problem into smaller subproblems [7], and BDLS an algorithm that splits the problem up into two different algorithms and determines the failure rate in an attempt to correct it efficiently [8], AIRSN [2], a grid based application previously used in neuroscience data analysis, LIGO [3], a virtual data grid for the Laser Interferometer Gravitational-wave Observatory (LIGO) that will greatly increase the accessibility of the vital data produced by potential gravitational waves, Montage [4], an application which is being repurposed to process large sections of sky through abstract workflows, and SDSS [5], a virtual database system designed to process large amounts of data and recently, the identification of galaxy clusters within the Sloan Digital Sky Survey.

Results from testing show clearly that BHEFT does have an advantage over some of the others in terms of successfully finding a suitable plan to process data. Another trend that can be seen is the success of BHEFT over others when decreasing the utilization rate, simulating the impact an existing load of resources can have upon intense big data workflows like these.

Possible future work that the researchers propose is the case of significant overestimations or underestimations of task execution time, rts in my implementation's case, and assessing its robustness during these times.

CONCLUSION

BHEFT is a very interesting and effective solution to the scheduling problem on cloud platforms. The work that this technical report describes barely scratches the surface into the world of scheduling algorithms let alone demonstrate a concrete example of its capabilities. I believe that BDC planning provides a very important and extremely useful idea and explanation of bicriteria, choosing to create efficiency with two variables at once. There will always be a line that we can approach but never get to in terms of optimization, especially in a field as complicated as big data processing.

The implementation of BHEFT into GCC went smoothly, now allowing users to use an easy to understand and navigable UI to explore the world of cloud computation. Despite the numerous obstacles that came while attempting to implement something as large and daunting as BHEFT, we somehow were able to pull it off successfully.

[1] Zheng, W., Sakellariou, R. Budget-Deadline Constrained Workflow Planning for Admission Control. *J Grid Computing* **11,** 633–651 (2013).

[2] Van Horn, J. D., et al. "Grid-Based Computing and the Future of Neuroscience Computation, Methods in Mind." (2005).

[3] E. Deelman *et al*., "GriPhyN and LIGO, building a virtual data Grid for gravitational wave scientists," *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, 2002, pp. 225-234, doi: 10.1109/HPDC.2002.1029922.

[4] Berriman, G. B., et al. "Montage: A grid enabled image mosaic service for the national virtual observatory." *Astronomical Data Analysis Software and Systems (ADASS) XIII*. Vol. 314. 2004.

[5] J. Annis, Yong Zhao, J. Voeckler, M. Wilde, S. Kent and I. Foster, "Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey," *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002, pp. 56-56, doi: 10.1109/SC.2002.10021.

[6] M. Wieczorek, S. Podlipnig, R. Prodan and T. Fahringer, "Bi-criteria Scheduling of Scientific Workflows for the Grid," *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2008, pp. 9-16, doi: 10.1109/CCGRID.2008.21.

[7] Sakellariou, Rizos, and Henan Zhao. "A hybrid heuristic for DAG scheduling on heterogeneous systems." *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*. IEEE, 2004.

[8] Doğan, Atakan, and Füsun Özgüner. "Biobjective scheduling algorithms for execution time–reliability trade-off in heterogeneous computing systems." *The Computer Journal* 48.3 (2005): 300-314.