اصلاحیه تبصره ۳ ماده ۵۹ آیین نامه استخدامی اعضای هیات علمی

دستور اول از یازدهمین نشست کمیسیون دائمي دوره ششم هيأت امناي دانشگاه تهران ۱۰ مرداد۱۳۹۶

مصوبه:«کمیسیون دائمی هیات امنا به استناد ماده ۱ قانون احکام دائمی برنامه های توسعه کشور، مصوب ۱۰/۱۱/۱۳۹۵ مجلس شورای اسلامی و بر اساس بند ۲ مصوبه مورخ ۴/۲/۱۳۹۶ هیات رئیسه دانشگاه با افزودن یک بند به جدول ضرایب فوق العاده مدیریت پست های مدیریتی تبصره ۳ ماده ۵۹ آیین نامه استخدامی اعضای هیات علمی به شرح زیر موافقت کرد:

تبصره ۳- ضریب فوق‌العاده مدیریت هر پست مدیریتی مصوب با توجه به حجم و سختی کار، بازدهی و کارایی، مرتبه علمی عضو و در مقایسه با سایر مدیران بر اساس دامنه ضرایب مندرج در جدول ذیل به وسیله رئیس دانشگاه تعیین می‌شود:

جدول حداقل و حداکثر ضرایب فوق‌العاده مدیریت پست‌های مدیریتی

ردیف	پست مدیریتی مصوب	حداقل و حداکثر ضریب فوق‌العاده مدیریتی
۱	معاونان دانشگاه	۶۰ تا ۸۰ درصد
۲	روسای مراکز و ادارات کل	۶۰ تا ۷۰ درصد
۳	مدیران دفاتر و ادارات و مدیریت‌های بلافصل رئیس دانشگاه	۵۰ تا ۶۰ درصد
۴	رؤسای دانشکده‌ها، پژوهشکده‌ها، آموزشکده‌ها، مراکز، کتابخانه مرکزی و مؤسسات وابسته به دانشگاه	۴۰ تا ۶۰ درصد
۵	مدیران دفاتر و ادارات و مدیریت‌های زیر مجموعه بلافصل معاونت‌های دانشگاه	۳۰ تا ۴۰ درصد
۶	معاونان مدیران دفاتر و ادارات و مدیریت‌های بلافصل رئیس دانشگاه	۴۰ تا ۵۰ درصد
۷	معاونان دانشکده‌ها، پژوهشکده‌ها، آموزشکده‌ها، مراکز و مؤسسات وابسته	۳۰ تا ۵۰ درصد

۸

مدیران گروه‌های آموزشی و پژوهشی

۲۰ تا ۴۰ درصد

۹

معاونان مدیران دفاتر و ادارات و مدیریت‌های زیر مجموعه بلافصل معاونت‌های دانشگاه

۲۰ تا ۴۰ درصد

۱۰

سایر پست‌های مدیریتی ستاره‌دار مصوب

بر اساس مفاد تبصره ۴ این ماده تعیین می‌شود

دستور دوم: آیین نامه تشکیل شورای بین الملل دانشگاه

متن کامل در فایل پیوست است

7caa5ccb

محمود نیلی احمدآبادی
رئیس دانشگاه تهران

and future call to process() yields

...his termination detection algo... which a thread t is checking for ..., while another thread, not yet ...ad concluded to be idle (line 16) ...aking up another thread may af... he recently woken up thread con... t possible for thread t to conclude ...aph is still being processed. This ...rmination check is not performed ... in the load array may change dur...k.

...y translating the pseudo code al... carefully manipulating the inter... ing the tool Spin. The used imple... d process is a breadth-first explo... sed graph is a chain of 10 nodes ...lidation test of the "old" termina... was that the algorithm contains ...gorithm has the flaw that when ...d call of process(), then the value ...ill that of the last call of process ...es a thread to evaluate outdated ...clusion that a thread is waiting, ...ing. This can cause problems due ...e of the execution of the method

...med problem can easily be over-

The new termination detection algorithm (listing 2) is very much the same as the old algorithm. In the new version, however, the problem with the non-atomic termination check has been solved by making it atomic. A single atomic integer is used instead of an array with boolean values (line 1). A thread that is waiting is represented by decreasing the value of that integer (line 11). When the value of working is evaluated to be zero (line 12), then all threads have concluded that there are no more tasks to perform, indicating that the calculations have finished (the next section will elaborate on these implications).

Every thread running the algorithm loops over the following sequence:

A call to (an implementation of) process() is performed. This returns an integer representation of the amount of tasks currently available (line 8). If there are still tasks left then the thread will set waiting to false, waking up all threads currently waiting (line 18).

When there are currently no tasks available then the following section of the algorithm is executed.

- The boolean waiting is set to true to keep threads waiting until more tasks become available. (line 10)
- The value of working is decreased by one to indicate that the thread is waiting. (line 11)
- working is evaluated. If it is equal to zero, then done is atomically set to true, indicating that the program should terminate, else it is left unchanged. (line 12)

done and every current and future call to process() yi... zero.

The main concern with this termination detection a...rithm is the situation in which a thread t is checking ... termination (lines 6-11), while another thread, not ... checked, wakes up a thread concluded to be idle (line ... In that case the thread waking up another thread may ...terwards go idle while the recently woken up thread c...tinues to work, making it possible for thread t to concl... termination while the graph is still being processed. T... may occur because the termination check is not perform... atomically and the data in the load array may change d...ing the termination check.

This flaw was exposed by translating the pseudo code ...gorithm to Promela and carefully manipulating the in...leaving of the threads using the tool Spin. The used im...mentation of the method process is a breadth-first ex...ration of a graph. The used graph is a chain of 10 no... The first result of the validation test of the "old" termi...tion detection algorithm was that the algorithm conta... at least one flaw. The algorithm has the flaw that w... a thread is in the method call of process(), then the v... of load[processorId] is still that of the last call of pro... (possibly zero). This causes a thread to evaluate outda... data and leads to a conclusion that a thread is wait... when it is in fact processing. This can cause problems ... to the non atomic nature of the execution of the met... process.

Luckily, the aforementioned problem can easily be o...

- The value of working is decreased by one to indicate that the thread is waiting. (line 11)
- working is evaluated. If it is equal to zero, then done is atomically set to true, indicating that the program should terminate, else it is left unchanged. (line 12)

data and leads to a conclusion that a thread is waiting, when it is in fact processing. This can cause problems due to the non-atomic nature of the execution of the method process.

Luckily, the aforementioned problem can easily be over...

- The value of working is dec... that the thread is waiting.
- working is evaluated. If it i... is atomically set to true, ind... should terminate, else it is ...

algorithm will be discussed in the related work section.

This algorithm can easily be transformed in a parallel algorithm by placing the code in while(!queue.empty) in the method process of the termination detection algorithm. This does however pose a problem when two vertices, both having an edge to the same node, execute the code result.push(endV) (line 15) simultaneously. This causes the vertex to be pushed to the queue multiple times. For this the construct listed in listing 4 has been devised.

```
//AtomicBoolean[] visited;
boolean vis = visited[endV.id].getAndSet(
    true);
if(!vis){
    queue.push(endV)
}
```

Listing 4. Adaptation for topological sort

With this construct multiple threads executing the code visited[endV.id].getAndSet(true) will only yield false once and only one thread will push endV to the queue.

The resulting algorithm is listed in listing 5.

```
1  Graph g;
2  Stack result = new Stack();
3  Stack queue = new Stack();
4  AtomicBoolean[] visited;
5
6  void init(Graph graph){
7      g = graph;
8      for(v in g.vertices){
9          if(v has no incoming edges){
10             visited[v.id].set(true)
11             queue.push(v);
```

Listing 5. Parallel topological sort

This algorithm works very much the same as the sequential version. However, now all threads operate on the shared queue and result stacks and the aforementioned control sequence is inserted to ensure that a vertex is not added multiple times. The code t.start() (line 15) will result in that thread calling the termination detection algorithm, which will repeatedly call doTopSortProcess() (line 28). After the processing has finished the threads will join (lines 16-18) and a check if the graph was acyclic (lines 21-24).

7.2 Correctness

The important properties of a parallel topological sort algorithm are that a vertex is added to the list once (and only once) and that the resulting list retains the topological order of the graph. The first property is guaranteed by the earlier discussed modification, displayed in listing 4. The second property also holds. This is because a vertex cannot be added to the list as long as it has incoming edges and the algorithm ensures that a vertex is added to the list before its edges are removed and the connected vertices can be inserted into the list (listing 5, lines 32-42).

This implementation also satisfies the requirements for an implementation of process(), set in section 7.2. A vertex has only a limited amount of outgoing edges, so the for-loop (line 33) will eventually break. The algorithm has no infinite loops and a thread running the algorithm is never paused, which means that the algorithm is deadlock free. The other two requirements are also met. Tasks are represented by a vertex in the queue, which can only in a certain direction. This interface has been devised by De Heus [4].

- **VertexIdGraph** contains the functionality to assign every vertex a unique identifier and retrieving the identifier of a vertex or the vertex with a certain identifier. This is a newly created interface.
- **TopologicalSortGraph** is an interface containing methods that are needed for the implemented topological sort algorithms. Mainly, removing an edge from a graph and telling whether a vertex has incoming edges.

Figure 1 provides an outline of how these interfaces extend each other.

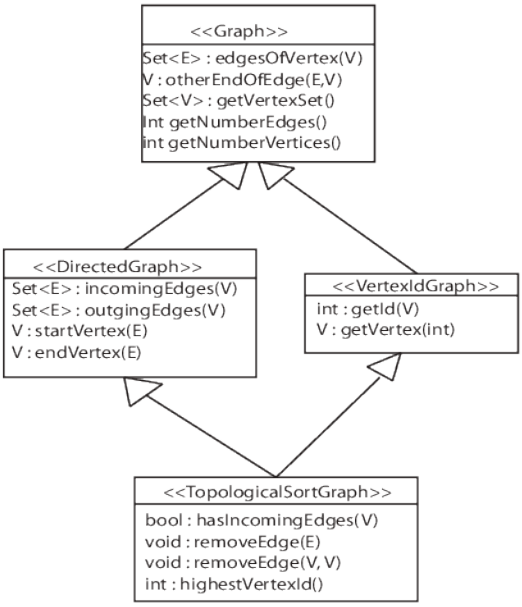

Figure 1. Interface structure

8. SPEED TEST

In this section the results of the speed tests of the sequential and parallel topological sort algorithms will be presented.

8.1 Setup

The speed tests were run by submitting the sequential and parallel algorithms to some (pseudo-)randomly generated graphs. The graphs were generated by repeatedly adding a vertex to the graph and, with certain possibility, adding an edge from every other vertex to that vertex. Whether an edge was actually added to the graph was decided by comparing a randomly generated value to a specified possibility threshold. The generated graphs each had a size of 500, 1000, 2000, 5000 or 10000 vertices and had a probability threshold. The probabilities that were used where 0.001, 0.005 and 0.01. The algorithms were run five times on each of the graphs and the average time, measured using the Java *System.nanotime()* method, was calculated.

The machine used for the tests had the following specifications:

- Processor: Intel Core i7-3770 CPU at 3.40GHz (4 cores)
- Memory: 4x 8192 MB DDR3 at 1333 MHz
- Chipset: MSI Z77A-GD65

8.2 Results

Table 1 displays the average run times for the algorithms to run on different graphs. The numbers on the top represent the complexity of the graph represented as the likelihood of an edge being added to the graph upon creation, the numbers on the left are size of the graphs expressed in the amount of vertices in the graph. For every complexity and size there are three values: The time (in milliseconds) it took the sequential DFS algorithm, the time it took the parallel algorithm with one thread and the time it took the parallel algorithm with four threads to solve the graph.

size/complex		0,001	0,005	0,01
500	DFS	2968	1225	803
	1 core	2907	3499	14988
	4 cores	102303	10177	101361
1000	DFS	665	401	980
	1 core	1032	1700	2427
	4 cores	101520	101914	102474
2000	DFS	580	2291	6117
	1 core	2121	5543	10072
	4 cores	830775	104574	107741
5000	DFS	13215	12252	23562
	1 core	17798	33051	74445
	4 cores	110676	120437	132351
10000	DFS	14660	48473	105947
	1 core	25591	129034	263600
	4 cores	114401	148720	205740

Table 1. Results of speed tests

8.3 Discussion

The complexity of the topological sort algorithm devised by Kahn, which was used for the parallel implementation has complexity $O(|V|+|E|)$. This complexity is the same as the complexity of the depth-first search topological sort algorithm. When the parallel part of the parallel algorithm is run in parallel it should have an approximate complexity of $O(\frac{|V|+|E|}{n})$, where n stands for the number of processors, a complexity lower than that of the DFS topological sort algorithm.

However the results of the speed tests contradict this premise. The runtime of every sequential DFS run on a graph was faster than the parallel version with four threads on four cores. This can be caused by a combination of factors:

Overhead. The termination detection used to control the parallel algorithm may introduce overhead that can cause the algorithm to run slower. The introduced overhead may be investigated by comparing the run times of the parallel algorithm, using one thread, with the pure, sequential implementation of Kahn's algorithm.

Different algorithms. The parallel algorithm is not an adaptation of the sequential DFS algorithm. It is a possibility that the algorithm devised by Kahn naturally runs slower than the DFS topological sort. This difference may be explored by comparing the run

Low task parallelism. When new tasks (in this case vertices) become available at a slow rate, then most of the threads will mostly be waiting. This diminishes the capacity to run in parallel and impacts the performance. This may be addressed by further parallelizing sub-tasks performed within the *doTopSort-Process()* method.

The performed tests conclude that the sequential algorithm is the superior algorithm for all the tested graphs. However they do show that as the graph becomes bigger, the proportional difference becomes smaller, as displayed in table 2. The displayed ratios are calculated by dividing the runtime of the parallel algorithm executed with four threads by runtime of the sequential algorithm.

size/complex	0,001	0,005	0,01
500	84,0	157,1	128,7
1000	153,8	254,8	104,5
2000	1432,4	45,7	17,6
5000	8,4	2,2	2,0
10000	7,9	3,0	1,9

Table 2. Speed ratios (4 cores/DFS)

With the exception of graph of size 1000 with complexity of 0,001 and 0,005, all proportional speed differences become smaller as the graph becomes bigger or more complex. This leads to believe that for a large enough graph, the parallel algorithm may be faster than the sequential one.

9. RELATED WORK

9.1 DFS topological sort

This algorithm is used as the sequential implementation of the topological sort algorithm, so that it might be compared in terms of speed. For understanding of the reader, the DFS topological sort algorithm is described below.

The sequential topological sort algorithm is a fairly plain adaptation of the depth-first search traversal of a graph. The nodes are visited via a post-order traversal. This algorithm was first described by Tarjan [12]. The pseudocode algorithm of the implemented sequential topological sort algorithm is listed below in listing 6.

```
        tempmark[v.id]=false;
        permmark[v.id]=true;
    }
}
```

Listing 6. Depth first search topological sort

The main method of the algorithm is the topSort(Graph g) method. It takes the graph to be sorted as an argument and the result is a stack containing the sorted graph or null when the graph is not a directed, acyclic graph. The processing algorithm is the processVertex(Vertex v) method. It works as follows.

- It takes a vertex as argument and recursively calls the processVertex method on its child vertices (lines 22-24), before adding v to the resulting stack (line 26). This results in a post-order ordering of the tree with v as root.
- Along the way the algorithm sets the values of tempmark and permmark, to mark a vertex as temporarily or permanently visited (lines 21-26).
- The tempmark array is used to detect a cycle in the graph (line 20), detecting that the graph is not a directed, acyclic graph and sets the value of isDAG to true.
- The permmark array is used to detect that the calculations on a vertex and its children has already been completed, so the algorithm can skip that node.

When the processing has finished, the thread in the main method checks if the isDAG flag has been set. When it has not, it returns a stack containing the topological ordering of the graph. When it does it returns null. [3]

9.2 M.C. Er

An alternative algorithm for topological sort was devised by M.C. Er [6]. The algorithm relies on value iteration to deduce a correct topological ordering of a graph. The algorithm starts at the nodes that have no incoming edges and then follows the edges, assigning every encountered vertex an integer value. With every vertex the integer value assigned is the value of the vertex visited before incremented by one. When a thread encounters a vertex that has al-