Illustration for XML Parallel Query Processing

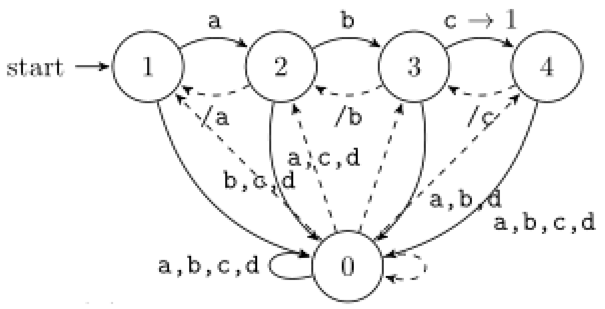
1 Examples for XML Query Processing using Parallel Pushdown Transducers

e.g.1

(1) XML sample

```
<a>
<b>
<d>xxx</d>
</b>
<b>
<c></c>
</b>
</a>
```

(2) Create automata for XPATH:/a/b/c



(3) Split the XML into two parts

Part 1

```
<a>
<b>
<d>xxx
```

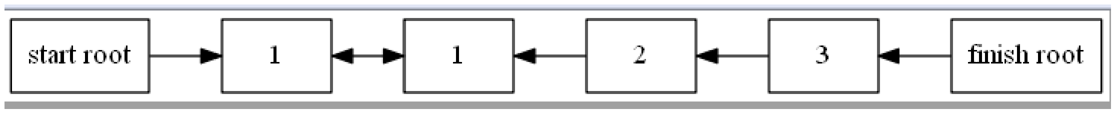
Part 2

```
</d>
</b>
<b>
<c></c>
</b>
</a>
```

(4) In parallel phase, create state mapping for each chunk

For Part 1

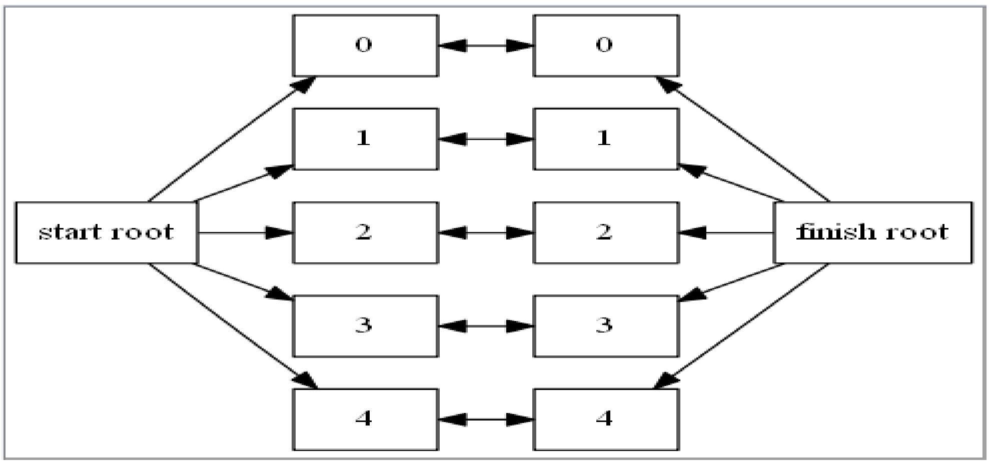
Push a,b,d into stack sequentially, get the tree which represents all the states and stacks in the mapping as follows:



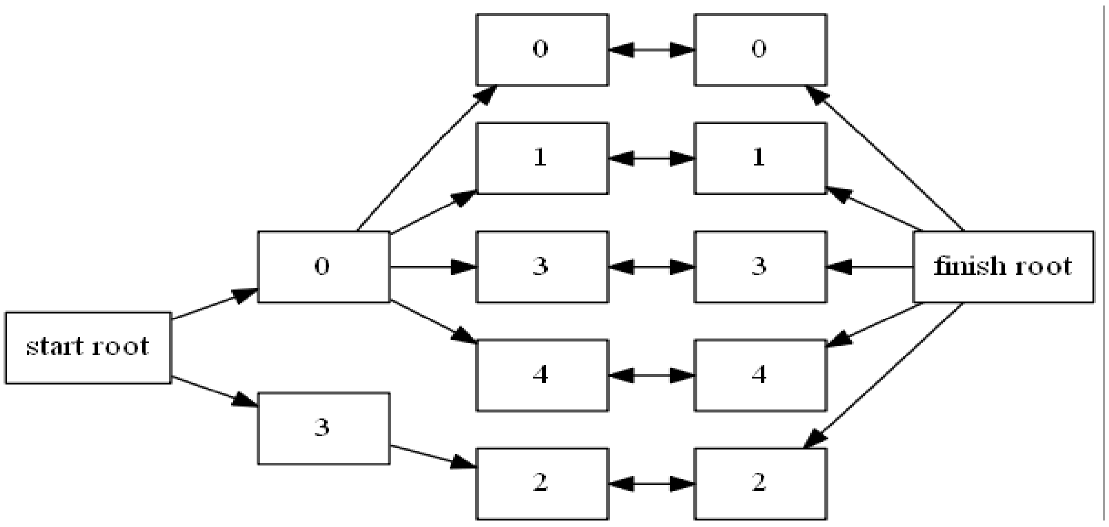
Therefore, the mapping for part1 is (1, ε)->(3, 2:1, ε).

For Part 2

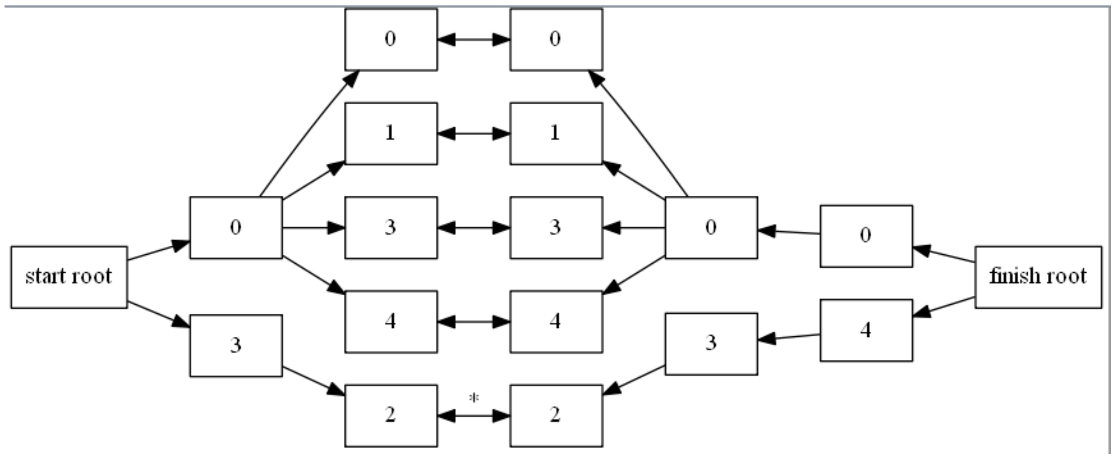
<1> get the starting state of the tree



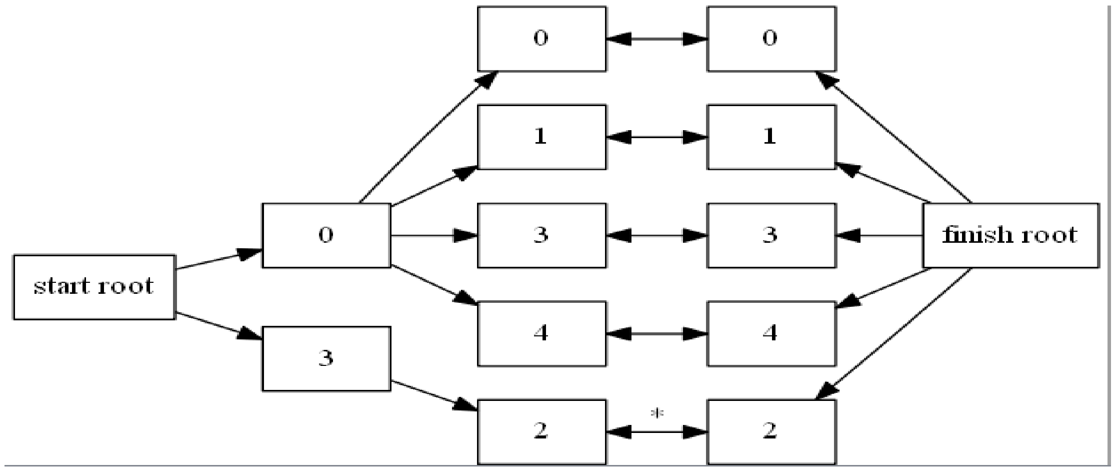
<2> pop b from the stack</b>



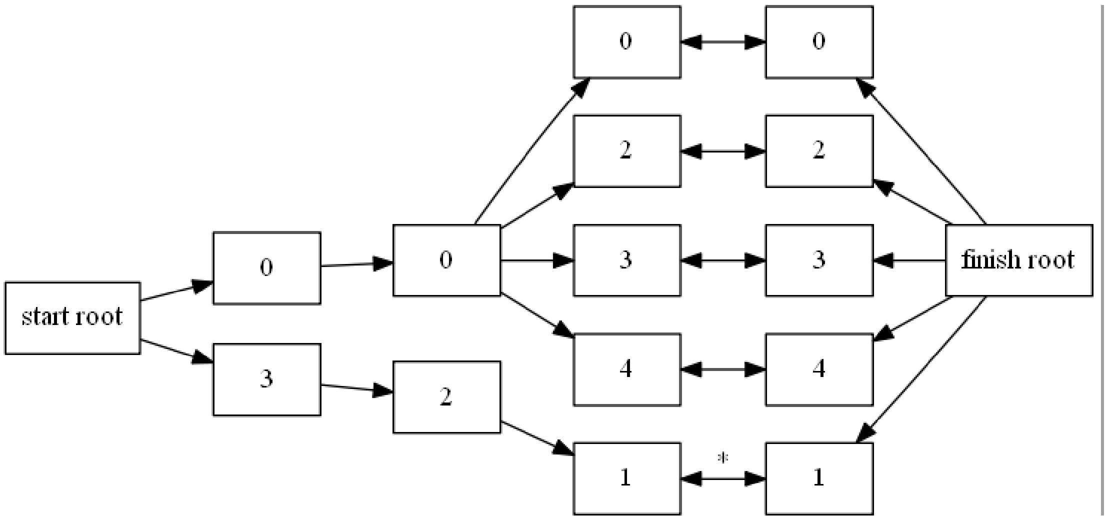
<3> push b,c into stack sequentially<b><c>



<4>pop c,b from stack sequentially</c></b>



<5>pop a from stack</a>

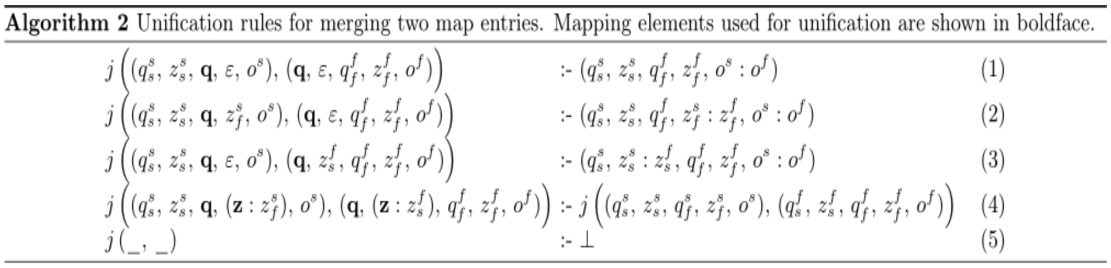


<6> combine the mappings for Part1&Part2

Part1 is (1, ε)->(3, 2:1, ε)

Part2 is (3, 2:1)->(1, ε, 1)

According to the rules2 in this paper, the result is (1, ε)-> (1, ε, 1) .



Algorithm 2 Unification rules for merging two map entries. Mapping elements used for unification are shown in boldface.

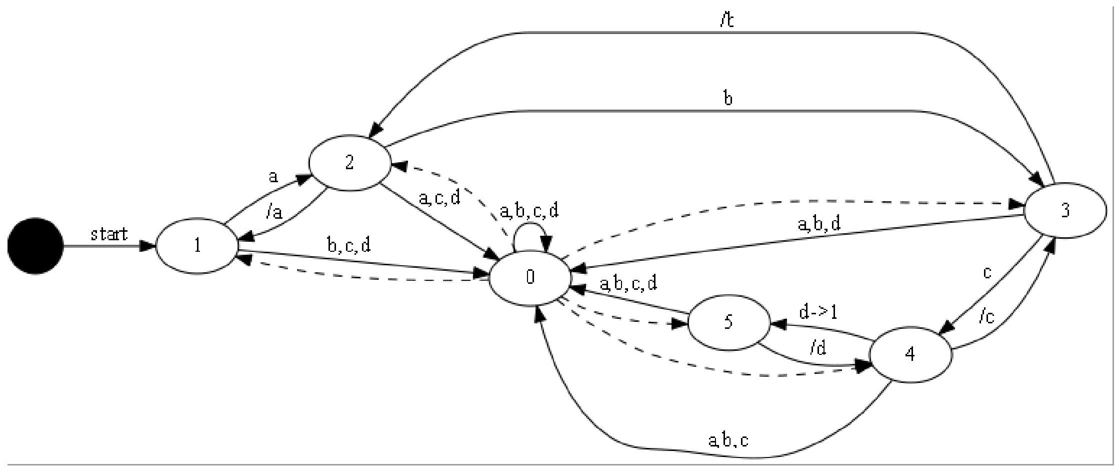
$$
\begin{array}{lll}
j\left((q_s^s, z_s^s, \mathbf{q}, \varepsilon, o^s), (\mathbf{q}, \varepsilon, q_f^f, z_f^f, o^f)\right) & \text{:- } (q_s^s, z_s^s, q_f^f, z_f^f, o^s : o^f) & (1) \\
j\left((q_s^s, z_s^s, \mathbf{q}, z_f^s, o^s), (\mathbf{q}, \varepsilon, q_f^f, z_f^f, o^f)\right) & \text{:- } (q_s^s, z_s^s, q_f^f, z_f^s : z_f^f, o^s : o^f) & (2) \\
j\left((q_s^s, z_s^s, \mathbf{q}, \varepsilon, o^s), (\mathbf{q}, z_s^f, q_f^f, z_f^f, o^f)\right) & \text{:- } (q_s^s, z_s^s : z_s^f, q_f^f, z_f^f, o^s : o^f) & (3) \\
j\left((q_s^s, z_s^s, \mathbf{q}, (\mathbf{z} : z_f^s), o^s), (\mathbf{q}, (\mathbf{z} : z_s^f), q_f^f, z_f^f, o^f)\right) & \text{:- } j\left((q_s^s, z_s^s, q_f^s, z_f^s, o^s), (q_s^f, z_s^f, q_f^f, z_f^f, o^f)\right) & (4) \\
j(\_, \_) & \text{:- } \bot & (5)
\end{array}
$$

e.g.2

(1) XML sample

```
<a>
<b>
<c>
<d>xxx</d>
</c>
</b>
</a>
```

(2) Create automata for XPATH:/a/b/c/d



(3) Split the XML into two parts

Part 1

```
<a>
<b>
<c >
```

Part 2

```
<d>xxx</d>
</c>
</b>
</a>
```

(4) In parallel phase, create state mapping for each chunk

For Part 1

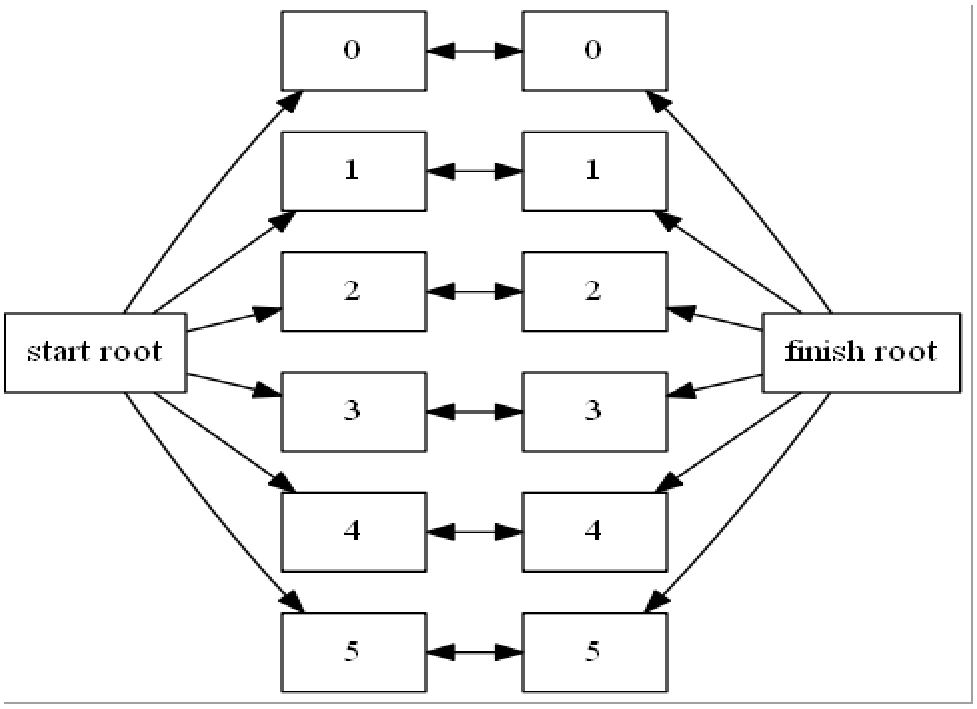
Push a,b,c into stack sequentially, get the tree which represents all the states and stacks in the mapping as follows:

$$\begin{tikzcd}
\boxed{\text{start root}} \arrow[r] & \boxed{1} \arrow[r, leftrightarrow] & \boxed{1} & \boxed{2} \arrow[l] & \boxed{3} \arrow[l] & \boxed{4} \arrow[l] & \boxed{\text{finish root}} \arrow[l]
\end{tikzcd}$$

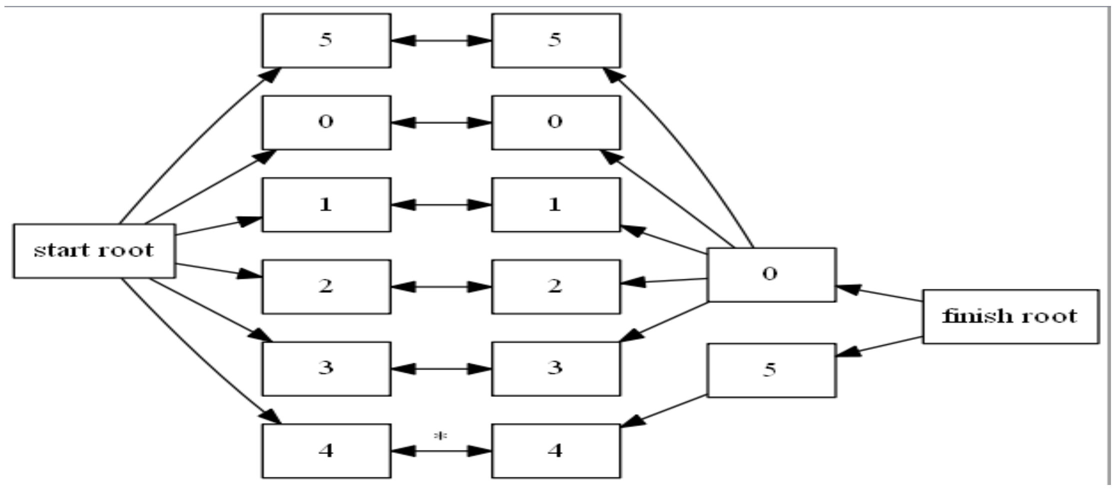
Therefore, the mapping for part1 is (1, ε)->(4, 3:2:1, ε).

For Part 2

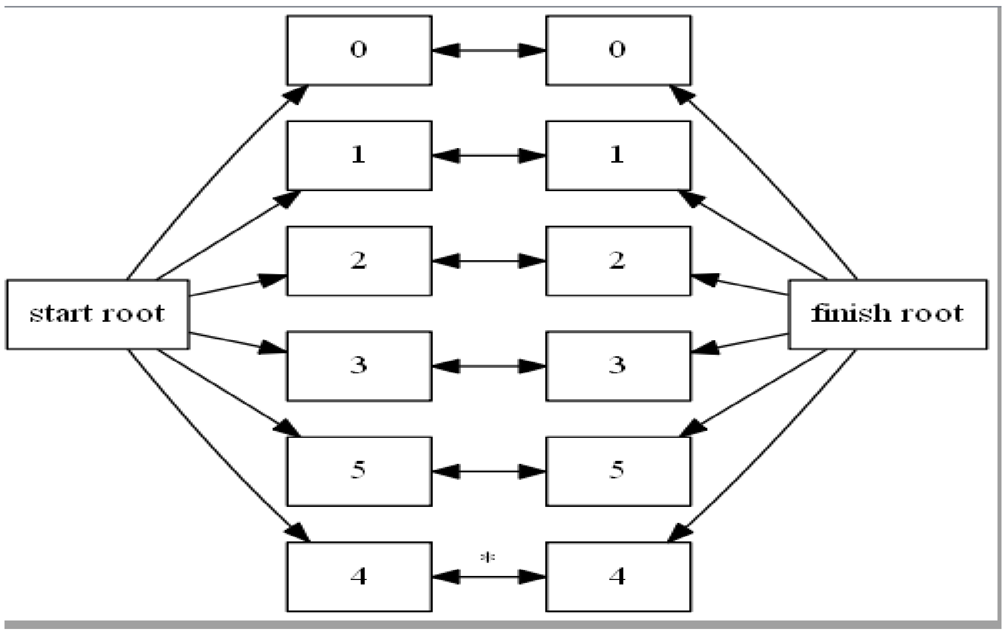
<1> get the starting state of the tree



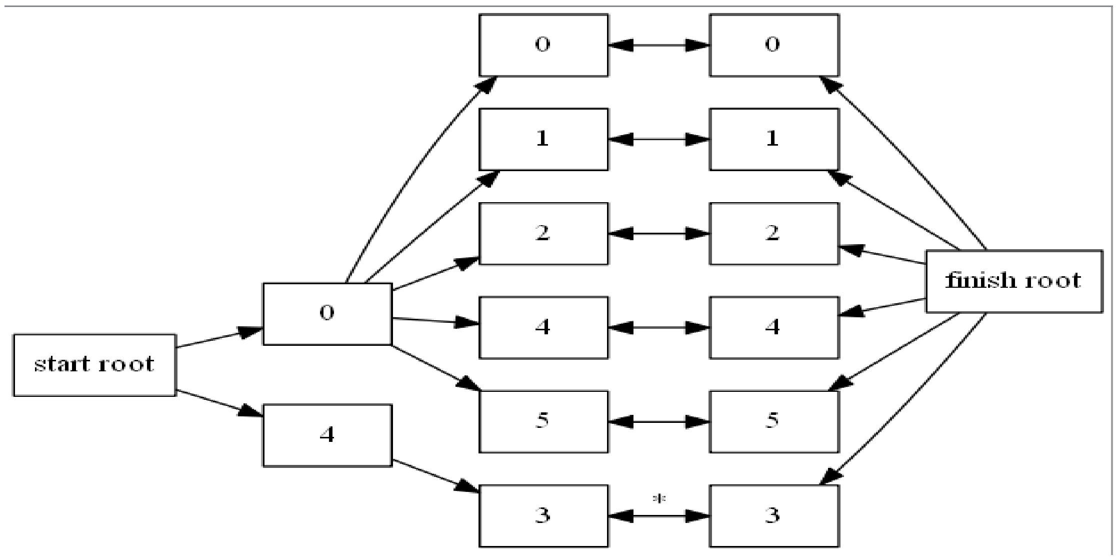
<2> push d into the stack<d>



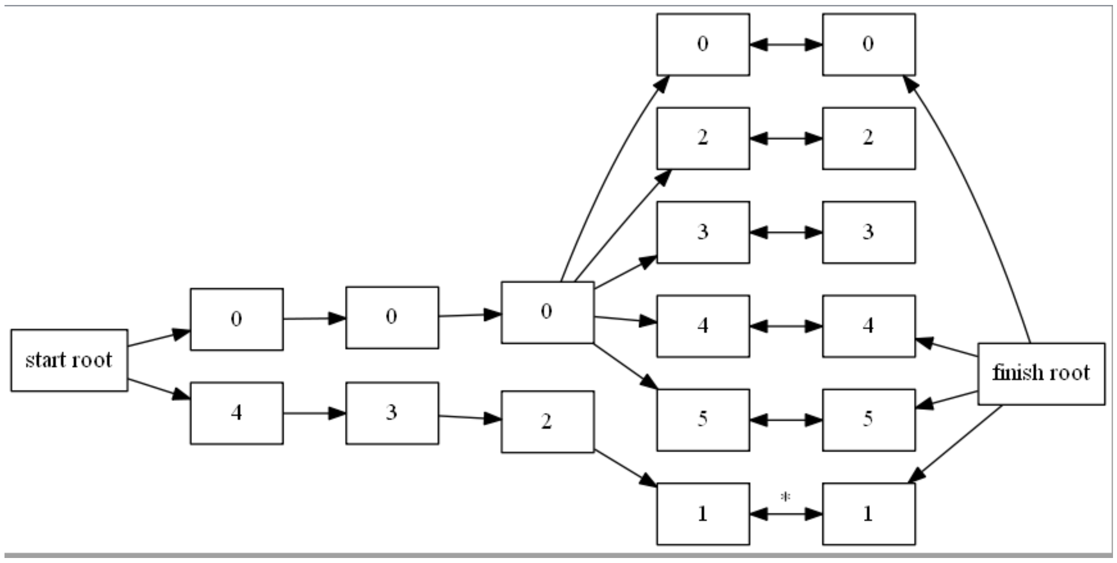
<3>pop d from stack</d>



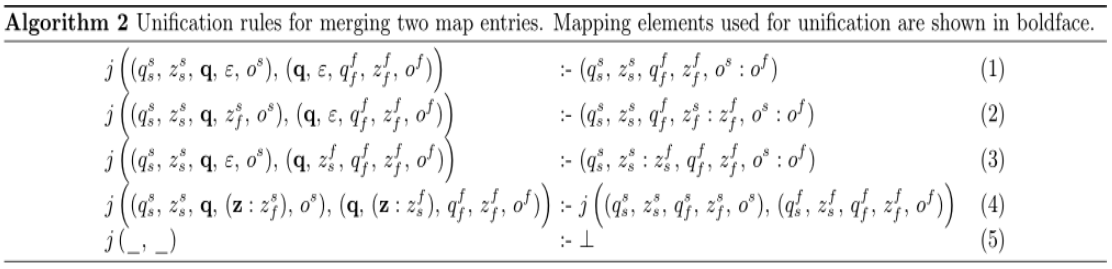
<4> pop c from stack</c>



<5>pop b,a from stack sequentially</b></a>



<6> combine the mappings for Part1&Part2
Part1 is (1, ε)->(4, 3:2:1, ε)
Part2 is (4, 3:2:1)->(1, ε, 1)
According to the rules4 in this paper, the result is (1, ε)-> (1, ε, 1) .



Algorithm 2 Unification rules for merging two map entries. Mapping elements used for unification are shown in boldface.

```
$j\left((q_s^s, z_s^s, \mathbf{q}, \varepsilon, o^s), (\mathbf{q}, \varepsilon, q_f^f, z_f^f, o^f)\right)$    :- $(q_s^s, z_s^s, q_f^f, z_f^f, o^s : o^f)$    (1)
$j\left((q_s^s, z_s^s, \mathbf{q}, z_f^s, o^s), (\mathbf{q}, \varepsilon, q_f^f, z_f^f, o^f)\right)$    :- $(q_s^s, z_s^s, q_f^f, z_f^s : z_f^f, o^s : o^f)$    (2)
$j\left((q_s^s, z_s^s, \mathbf{q}, \varepsilon, o^s), (\mathbf{q}, z_s^f, q_f^f, z_f^f, o^f)\right)$    :- $(q_s^s, z_s^s : z_s^f, q_f^f, z_f^f, o^s : o^f)$    (3)
$j\left((q_s^s, z_s^s, \mathbf{q}, (\mathbf{z} : z_f^s), o^s), (\mathbf{q}, (\mathbf{z} : z_s^f), q_f^f, z_f^f, o^f)\right)$    :- $j\left((q_s^s, z_s^s, q_f^s, z_f^s, o^s), (q_s^f, z_s^f, q_f^f, z_f^f, o^f)\right)$    (4)
$j(\_, \_)$    :- $\perp$    (5)
```

2 Some significant functions for the source code of XML Parallel Query Processing

(1) void createAutoMachine(char* xmlPath);
Note: create an automata by the XPath Query command.
Parameter: xmlPath-- XPath Query command.
(2) int split_file(char* file_name, int n);
Note: this function can split a large file into several parts according to the number of threads for the program, while keeping the split XML files into the memory. Besides, when splitting XML files, the program could skip the default size to look for the next open angle bracket, and then form a string array which could be processed by many threads simultaneously.
Parameter: file_name—the name for the xml file; n—the number of threads for the program.
(3) void *main_thread(void *arg);
Note: the function would be called by each thread, using the following functions to deal with the related elements.
(4) void createTree_first(int start_state);
Note: initiate a stack tree for the first thread, the start_state is 1. This tree is used for dealing with the XPath Query command.
(5) void createTree(int thread_num);
Note: initiate a stack tree for other thread other than the first thread. This tree is used for dealing with the XPath Query command.
(6) int xml_process(xml_Text *pText, xml_Token *pToken, int multilineExp, int multilineCDATA, int thread_num);
Note: the function could be called by each thread, dealing with each line of the file. Besides, this function could identify the following elements, which include XML head, Start Tag(e.g <xxx>), End Tag(e.g </xxx>), Tag(e.g <xxx/>), Content for the Tag, XML Explanation, Attribute Name for Tag, Attribute Value for Tag, Content for CDATA element. Each element would be processed according to its type.
Parameter: pText—the content of the xml file; pToken—the type of the current xml element; multilineExp—whether the current line of the xml file is the multiline

explanation; multilineCDATA-- whether the current line of the xml file is the multiline CDATA; thread_num—the number of the thread.

(7) void push(Node* node, Node* root, int nextState);

Note: if type of the xml element is Start Tag(e.g <xxx>) and the content of the tag could be found in the automata, then the function which could be called by (6) would create a new node for the next state and push it into the stack tree.

Parameter: node-->the current node of the tree; root-->the root of the tree; nextState-->the state for the next node which would be pushed on top of the current node.

(8) void pop(char * str, Node* root)

Note: if type of the xml element is End Tag(e.g </xxx>) and the content of the tag could be found in the automata, then this function which could be called by (6) would delete the related node from the finishing stack tree. If no such node exists, a new node is created in the start tree, thus pushing the next state on the starting stack tree.

Parameter: str—the content of the xml element; root—the root of the tree.

(9) void add_node(Node* node, Node* root);

Note: add a node into the tree. Each tree node has at most one child for each state. If a transition causes two child nodes to have the same symbol, then two nodes would be merged. This function would be called by (7)&(8).

Parameter: node-->the current node would be added into the tree; root—the root of the tree.

(10) void thread_wait(int n);

Note: waiting for all the threads get ended.

(11) ResultSet getresult(int n);

Note: after all the threads get done, this function would be called to get all the mappings for the stack tree of the related thread, then merged them into one final mapping.

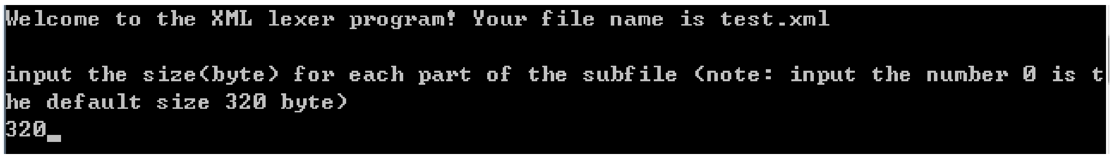
Parameter: n—total number for all the threads.

3 A result sample for the program running

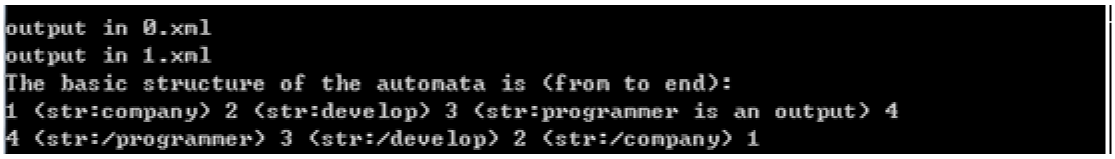
(1) Original xml file:

```
<?xml version="1.0" encoding="gb2312" ?>
  <!--employee's personal in
formation-->
<company name="Que's C++ studio">
        <sales>
                <salesman age="28" level="1" sex="male">Jack</salesman> <!--salesman
information-->
        </sales>
        <develop>
                <programmer age="35" sex="famale">Jane</programmer>
                <manager/><engineer age="40" sex="male" level="2">Johnson</engineer>
                <![CDATA[<hello><world><city>]]>
        </develop><!--member information-->
        <![CDATA[
                <hello>
                                <world>

                ]]>
</company>
```
(2) XPath: "/company/develop/programmer"

(3) Input the size of each part of the file, and create automata for XPath



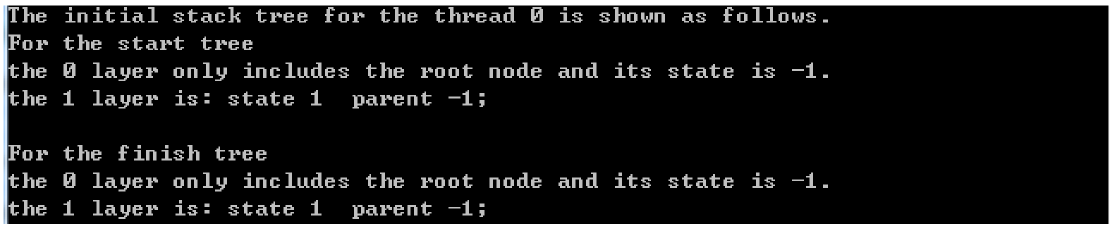
```
Welcome to the XML lexer program! Your file name is test.xml

input the size(byte) for each part of the subfile (note: input the number 0 is t
he default size 320 byte)
320_
```



```
output in 0.xml
output in 1.xml
The basic structure of the automata is (from to end):
1 (str:company) 2 (str:develop) 3 (str:programmer is an output) 4
4 (str:/programmer) 3 (str:/develop) 2 (str:/company) 1
```

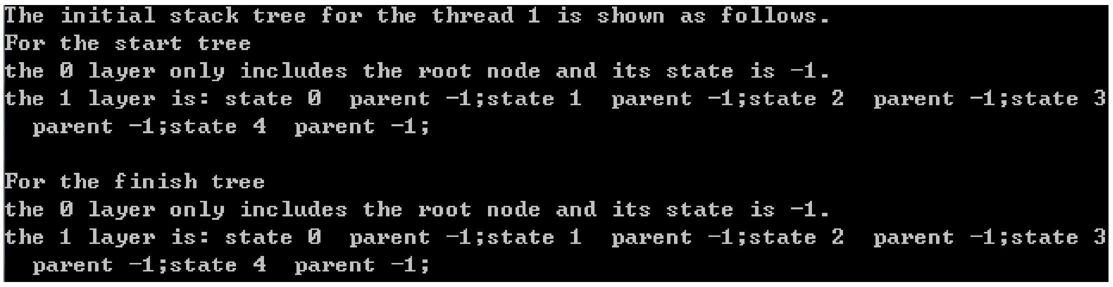
(4) Split the XML into two parts

Part1:
```
<?xml version="1.0" encoding="gb2312" ?>
  <!--employee's personal in
formation-->
<company name="Que's C++ studio">
        <sales>
                <salesman  age="28"  level="1"  sex="male">Jack</salesman>  <!--salesman
information-->
        </sales>
        <develop>
                <programmer age="35" sex="famale">Jane</programmer>
```
Part2:
```
        <manager/><engineer age="40" sex="male" level="2">Johnson</engineer>
                <![CDATA[<hello><world><city>]]>
        </develop><!--member information-->
        <![CDATA[
                <hello>
                                <world>

                ]]>
</company>
```
(5) For Part1, the initial stack tree is created.



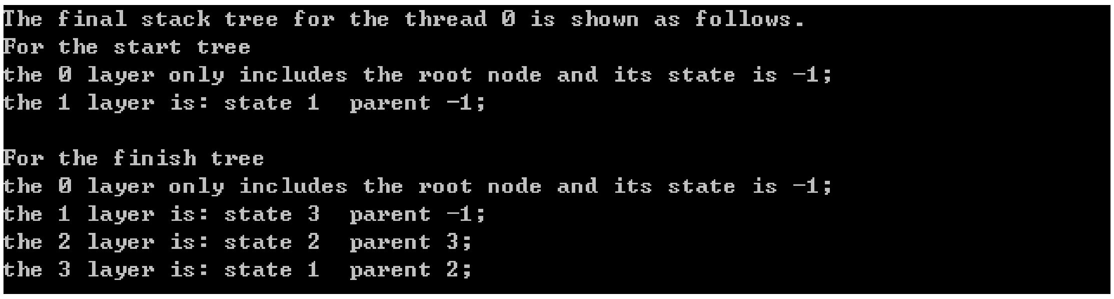
```
The initial stack tree for the thread 0 is shown as follows.
For the start tree
the 0 layer only includes the root node and its state is -1.
the 1 layer is: state 1  parent -1;

For the finish tree
the 0 layer only includes the root node and its state is -1.
the 1 layer is: state 1  parent -1;
```

(6) For Part2, the initial stack tree is created.



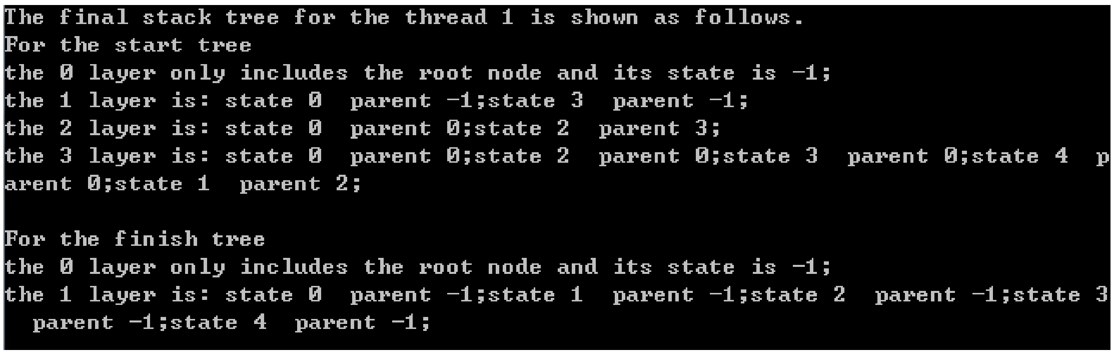
```
The initial stack tree for the thread 1 is shown as follows.
For the start tree
the 0 layer only includes the root node and its state is -1.
the 1 layer is: state 0  parent -1;state 1  parent -1;state 2  parent -1;state 3
  parent -1;state 4  parent -1;

For the finish tree
the 0 layer only includes the root node and its state is -1.
the 1 layer is: state 0  parent -1;state 1  parent -1;state 2  parent -1;state 3
  parent -1;state 4  parent -1;
```

(7) For Part1, get the final stack tree.



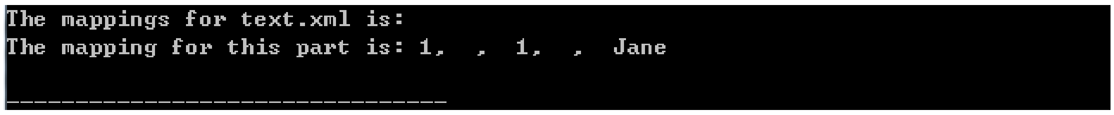
```
The final stack tree for the thread 0 is shown as follows.
For the start tree
the 0 layer only includes the root node and its state is -1;
the 1 layer is: state 1  parent -1;

For the finish tree
the 0 layer only includes the root node and its state is -1;
the 1 layer is: state 3  parent -1;
the 2 layer is: state 2  parent 3;
the 3 layer is: state 1  parent 2;
```

(8) For Part2, get the final stack tree.



```
The final stack tree for the thread 1 is shown as follows.
For the start tree
the 0 layer only includes the root node and its state is -1;
the 1 layer is: state 0  parent -1;state 3  parent -1;
the 2 layer is: state 0  parent 0;state 2  parent 3;
the 3 layer is: state 0  parent 0;state 2  parent 0;state 3  parent 0;state 4  p
arent 0;state 1  parent 2;

For the finish tree
the 0 layer only includes the root node and its state is -1;
the 1 layer is: state 0  parent -1;state 1  parent -1;state 2  parent -1;state 3
  parent -1;state 4  parent -1;
```

(9) Combine the mappings for Part1&Part2.



```
The mappings for text.xml is:
The mapping for this part is: 1,  ,  1,  ,  Jane

--------------------------------
```

4 other illustrations

(1) The program uses some source code from last assignment, which could identify some elements for a XML document, such as XML head, Start Tag(e.g <xxx>), End Tag(e.g </xxx>), Tag(e.g <xxx>), Content for the Tag, XML Explanation, Attribute Name for Tag, Attribute Value for Tag, Content for CDATA element.

(2) In order to run the code, for windows version, you can put an XML file named *test.xml* and a txt file named *XPath.txt* in the same folder with the *XML_parellel.exe*.

The XML_parellel.exe is in the folder called XML_CODE, and it can be executed directly by double clicking itself. Besides, two test xml files and source code are also put under the same folder with the *XML_parellel.exe*. Also, you need to input the number of bytes for each part of the file when the program is running.

(3) For Linux version, in order to run the program, you can put an XML file named *test.xml* and a txt file named *XPath.txt* in the same folder with the *XML_parellel*. Then transfer this folder to Linux server, enter into the related path and input the command to run this program.

```
./XML_parallel
```

(4) In order to compile the code, first you need to install a GCC compiling environment. Then open the command window and enter into the folder where the source code is in, input the following command to finish the process of compiling(it is also suitable for Linux environment):

```
gcc -O3 -o XML_parellel XML_parellel.c -lpthread
```

After that, you need to put a XML file named *test.xml* in the same folder with the source code. Then input the command XML_parellel directly in the same command window or double click *XML_parellel.exe* under the same folder and see the running results.

5 performance test results

This parallel program is tested on some large datasets to show its efficiency compare to the sequential version. The results are listed as follows.

(1)The configuration for the PC device is 2.1 GHz, Intel(R) Core(TM) i3-2310M CPU with 2 Core(with 4 threads), 4 GB of RAM.

(2)The first dataset is 226.725MB. The number of XML tags is 6503886, the number of match tags is 561, the number of output text element is 280.

Thread number	Splitting file(s)	Dealing with elements(s)	Merging results(s)	Total time(s)
sequential version	0.156	2.387	0.000	2.543
1	0.156	2.402	0.000	2.558
2	0.265	1.264	0.000	1.529
3	0.265	1.217	0.000	1.482
4	0.281	1.186	0.000	1.467
5	0.296	1.186	0.000	1.482
6	0.281	1.154	0.000	1.435
7	0.281	1.154	0.000	1.435
8	0.296	1.186	0.000	1.482

(3) The second dataset is 411.135MB. The number of XML tags is 11967168, the number of match tags is 1041, the number of output text element is 520.

Thread number	Splitting file(s)	Dealing with elements(s)	Merging results(s)	Total time(s)
sequential version	0.265	4.321	0.016	4.602
1	0.281	4.368	0.016	4.665
2	0.421	2.527	0.016	2.964
3	0.452	2.293	0.016	2.761
4	0.468	2.106	0.016	2.590
5	0.483	2.152	0.016	2.651
6	0.514	2.137	0.016	2.667
7	0.483	2.106	0.016	2.605
8	0.514	2.106	0.016	2.636

(4) The third dataset is 534.194MB. The number of XML tags is 15608820, the number of match tags is 1281, the number of output text element is 640.

Thread number	Splitting file(s)	Dealing with elements(s)	Merging results(s)	Total time(s)
sequential version	0.343	5.647	0.016	6.006
1	0.359	5.663	0.016	6.038
2	0.577	3.058	0.016	3.651
3	0.577	2.824	0.016	3.417
4	0.593	2.730	0.016	3.339
5	0.608	2.902	0.016	3.526
6	0.624	2.808	0.016	3.448
7	0.671	2.699	0.016	3.386
8	0.640	2.730	0.016	3.386

(5) From these data above, we can see the parallel version use less time than the sequential version, especially when dealing with elements. In this phase, this program initiate muti-threads to deal with each part of the file, and the time for this phase is about half of the sequential version. However, although the PC has 2 cores with 4 threads, when the number of threads is larger than 2, this new algorithm could not generate the same effect as 4 cores do since the PC has hyper-threads. Besides, the other two phases, splitting file and merging results, are executed sequentially. In this case, this algorithm is somewhat effective in XML processing.