HW 3: List Ranking and Tree Rooting

Course	ENEE651/CMSC751
Title	List Ranking and Tree Rooting
Date Assigned	Mar 27 2014
Date Due	ICE: April 4 11:59pm. XMTC: April 10 11:59pm
Contact	James Edwards

1. Assignment Goal

Identify The direction of the edges of a tree T(V,E) and a root r using the Euler tour and pointer jumping algorithms discussed in sections 9.1 and 9.2 respectively.

2. Problem Statement

Given a tree T(V, E) and some specified vertex r ϵ V, where V is the set of vertices and E the set of edges. The problem is to select a direction for each edge in E such that the resulting directed graph T'(V, E') is a (directed) rooted tree whose root is vertex r. Namely, all the edges are directed towards the root.

Brief algorithm description: you will need to perform Euler tour algorithm from section 9.1. For list ranking step, you will use the pointer-jumping algorithm from section 9.2.

3. Hints and Remarks

Separating concurrent reads and writes: Consider the pointer-jumping algorithm that you will use for list ranking step. In this step, for each edge (u, v), we will do an in-place update for all edges in parallel (Next[e] = Next [Next[e]]). This means that it will read 'next' for the next edge, to update its own. The same thing goes for the distance calculation. In a PRAM algorithm, the execution proceeds in synchronous manner, where first all processors read Next[Next[e]] before any of them write Next[e].

ICE is a language that follows the lock-step execution model of PRAM, where **all reads are done before any of the writes**, and no instruction will be executed by any parallel context until the previous instruction has completed execution completely on all parallel contexts.

The XMT platform, on other hand, implements a less-synchronous PRAM platform where the order in which the TCUs execute the above assignment is not determined. This can result in a mix of concurrent reads and writes to the elements of the arrays Next and Distance. Depending on the implementation of the memory read and write operations, this can cause the pointer graph to be left in an inconsistent or invalid state.

To avoid this issue, we propose the following scheme for your XMTC, use two arrays to store the pointer graph, e.g. Next_read and Next_write; perform all the read operations from the first array and all the write operations into the second one. For example, the above assignment can be rewritten as: Next_write[Next_read[i]] = Next_read[i]. Note that you need to ensure that the updated pointer graph is stored in the appropriate array at the end of each iteration.

4. Assignment

1. Write an ICE implementation of the tree-rooting algorithm using Euler tours and pointer jumping for ranking. Name your code rooting.ice.cpp.
2. Write an XMTC implementation of the tree-rooting algorithm using Euler tours and pointer jumping for ranking. Name your code rooting.c.

Note: a binary implementation of the serial algorithm will be provided with the files given to you. You are not required to do a serial implementation of this problem.

4.1. Setting up the environment

To get the source file templates and the Makefile for compiling programs, log in to your account in the class server and extract the rooting.tgz using the following command:

```
$   cp   /opt/xmt/class/xmtdata/rooting.tgz ~
$   tar  xzvf   rooting.tgz
```

This will create the directory *rooting* which contains the C file templates that you are supposed to edit, a C file for checking correctness and a Makefile.

Data files are located at a common location in the server *(/opt/xmt/class/xmtdata/rooting*). If you use the Makefile system explained in Section 4.4, you will not need to explicitly refer to this location. The provided Makefile utilizes command line options to pass the paths to the header and data files to the compiler.

4.2. Input Format

The Input is provided as the following:

#define N	The number of vertices in the tree
#define M	The number of edges in the tree (each edge counts twice)
#define NIL	This is the null node, its value is -1
Int root	The root vertex ID
int E[M][2]	The start and end vertex of each edge. Edges are provided as incidence list
int V[N]	The index in the edges array, where the edges incident to the vertex begin
int deg[N]	The degree of each vertex
int ptr[M]	The indices of the corresponding antiparallel edge
Int used[M]	Result array: The edges that are picked in the end

Declaration of temporary/auxiliary arrays: You can declare any number of global arrays and variables in your program as needed. For example, this is valid XMTC and ICE code:

```
#define T 16384
int temp1[16384];
int temp2[2*T];
int main() {
  //...
}
```

4.3. Data Sets

The following data sets are provided:

Dataset	N	M	Header file	Binary File
t1	64	126	$DATA/t1/rooting.h	$DATA/t1/rooting.xbo
t2	1024	2046	$DATA/t2/rooting.h	$DATA/t2/rooting.xbo
t3	32768	65534	$DATA/t3/rooting.h	$DATA/t3/rooting.xbo

$DATA is */opt/xmt/class/xmtdata/rooting*. Note that each edge is listed twice in the input file. For example, the undirected tree t1 has only 63 edges. A data set can be chosen by passing a DATA argument to the Makefile. See Section 4.4 for examples. You will still need to provide the entire path when using them with ICE.

4.4. Compiling and Executing

For your convenience, a Makefile is provided with the homework distribution. You can use the provided makefile system to compile and run your XMTC programs. To run the parallel rooting on the t1 data set, use the following command in the src directory:

```
> make run INPUT=rooting.p.c DATA=t1
```

This command will compile and run the rooting.p.c program with the t1 data set. For other programs and data sets, change the name of the input file and the data set. If you need to just compile the input file (no run):

```
> make compile INPUT=rooting.p.c DATA=t1
```

You can get help on available commands with

```
> make help
```

Note that, you can still use the xmtcc and xmtfpga commands as in the earlier assignments. You can run with the makefile system first to see the commands and copy them to command line to run manually. In case of the example we used above, the commands will look like:

```
> xmtcc -include ${DPTH}/t1/rooting.h ${DPTH}/d1/rooting.xbo rooting.p.c -o rooting.p
```

Where $DPTH is defined as /opt/xmt/class/xmtdata/rooting. If the program compiles correctly a file called rooting.p.b will be created. This is the executable you will run on the FPGA using the following command:

```
> xmtfpga rooting.p.b
```

Please use the ICE.py script to compile your ICE program. So in case of the example used above, the command will look like this:

```
> ICE.py -Include ${DPTH}/d1/rooting.h ${DPTH}/d1/rooting.xbo -i rooting.ice.cpp -o rooting.p
```

If successful, the binary executable rooting.p.b will be created. You can run that on the FPGA using the same command above.

5. Output

The array 'used' will have the edges used indicated with a value of 1.

Prepare and fill the following table: Create a text file named table.txt in doc**. Remove any printf statements from your code while taking these measurements**. Printf statements increase the clock count. Therefore the measurements with printf statements may not reflect the actual time and work done.

Dataset	**t1**	**t2**	**t3**
Parallel tree rooting clock cycles			
Serial tree rooting clock cycles			

Note that a part of your grading criteria is the performance of your parallel implementation on the largest dataset (t3). Therefore you should try to obtain the fastest running parallel program. As a guideline, for the larger dataset (t3) our Serial tree rooting runs in 270489217 cycles, and our Parallel Sample runs in 13647910 cycles (speedup ~19.8x) on the FPGA computer.

For this homework you do not need to provide the analysis for the ICE implementation, just your parallel ICE program.

Submission: The use of the make utility for submission '*make submit*' is required. Make sure that you have the correct files at correct locations (src and doc directories) using the make submitcheck command. Run following commands to submit the assignment:

```
$ make submitcheck
$ make submit
```