HW 6A: Graph Connectivity and Spanning Forests

Course:	ENEE759K/CMSC751
Title:	Graph Connectivity and Spanning Forests
Date Assigned:	April 22, 2010
Date Due:	May 11, 2010 11:59pm
Contact:	James Edwards – jedward5@umd.edu

1 Assignment Goal

Identify the connected components and a (not necessarily minimal) spanning forest for an undirected graph $G = (E, V)$ using the **second connectivity algorithm** described in section 11.2 of the class notes.

2 Problem Statement

Given an undirected graph $G = (V, E)$, the **second connectivity algorithm** identifies the connected components of G using $\log(n)$-proportional iterations. Implement an algorithm to derive a (not necessarily minimal) **spanning forest** from this algorithm.

Brief algorithm description. The definition of a *spanning forest* is given in Section 11.3 of the class notes. To derive a (not necessarily minimal) spanning forest, you can record all the edges which are used in the hooking steps of the Connectivity algorithm. More specifically, in the second connectivity algorithm, you will need to record the edges used in the second and third step of the algorithm, as described on page 89 of the class notes, namely the *Hooking on smaller* and *Hooking non-hooked-upon* steps.
Note. The order in which the edges in the spanning forest are discovered and recorded is arbitrary. In fact, for the same input graph, you might get different (correct) spanning forests at different runs.

The graph G is provided using the incidence list representation. See Figure 1.

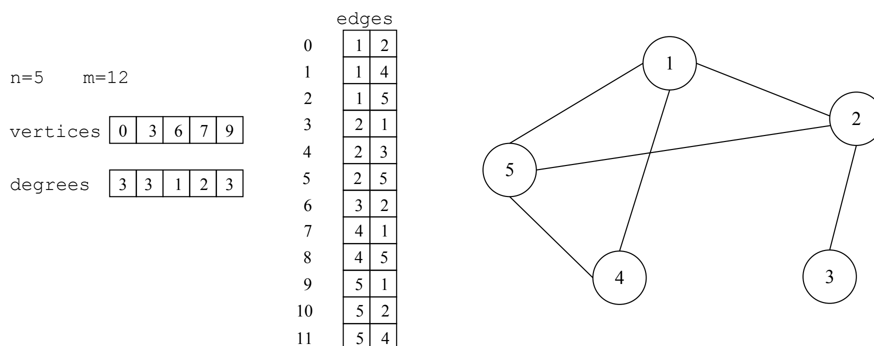

Figure 1: Incidence lists representation

3 Assignment

1. Implement the parallel algorithm for generating a spanning forest using the **Second Connectivity Algorithm**. Name your code file connectivity.p.c.
2. Implement a serial algorithm for generating a spanning forest using a serial connectivity algorithm of your choice. Name your code file connectivity.s.c.

Important: Both parallel and serial algorithms must be as efficient as possible. You will be graded based on both correctness and efficiency of your implementations.

4 Input

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 *connectivity.tgz* using the following command:

```
$ tar xzvf /opt/xmt/class10/xmtdata/connectivity.tgz .
```

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

As opposed to the previous assignments, data files are not included in this package. Instead they are located under */opt/xmt/class10/xmtdata/connectivity*. The Makefile system will automatically use the appropriate data files following your make commands so you don't need to make a copy of them in your work environment. You have the OS privileges to view the header files however you cannot edit them.

4.2 Input Format

The type and size of the data structures provided is given in the following table.

#define N	The number of vertices in the graph
#define M	The number of edges in the graph (each edge counts twice)
int edges[M][2]	The start and end vertex of each edge
int vertices[N]	The index in the `edges` array, at which point the edges incident to vertex begin
int degrees[N]	The degree of each vertex
int D[N]	Result array: stores the result pointer graph
int spanforest[N-1][2]	Result array: stores the edges in the spanning forest
int spanforest_size	Result value: gives the number of edges in the spanning forest

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 code:

```
#define N 16384

int temp1[16384];
int temp2[2*N];
int pointer;

int main() {
 //...
}
```

4.3 Data Sets

The following data sets are provided:

Dataset	N	M	Header file	Binary File
graph0	19	36	$DATA/graph0/connectivity.h	$DATA/graph0/connectivity.xbo
graph1	50	400	$DATA/graph1/connectivity.h	$DATA/graph1/connectivity.xbo
graph2	10000	40000	$DATA/graph2/connectivity.h	$DATA/graph2/connectivity.xbo
graph3	20000	40000	$DATA/graph3/connectivity.h	$DATA/graph3/connectivity.xbo

$DATA is */opt/xmt/class/xmtdata/connectivity*. Note that each edge is listed twice in the input file. For example, the undirected graph0 has only 18 edges.

Graph example. We have provided an example graph given by the dataset `graph0` for debugging and exemplification purposes. This corresponds to the graph in Figure 2.

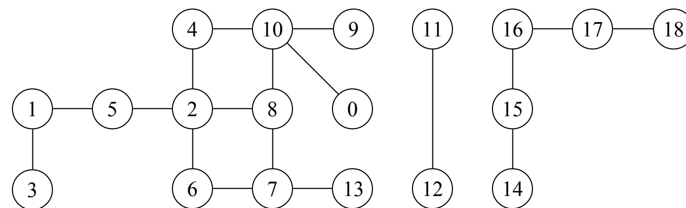

Figure 2: Graph example: `graph0` dataset.

4.4 Hints and Remarks

Separating concurrent reads and writes: Consider the first iteration of the algorithm applied to the graph in Figure 2. Initially, nodes 14-17 each belong to a trivial rooted star that consists of only one node, as shown in Figure 3.a.

In the *Hooking on smaller* step of the first iteration, the edges $\{(15,14),(16,15),(17,16)\}$ will be used to hook star 15 onto 14, star 16 onto 15 and star 17 onto 16 respectively. This is done by executing in parallel $D[D[u]] = D[v]$ for each edge (u,v) in the above set. In the PRAM algorithm, the execution proceeds in a synchronous manner, where first all the processors read $D[v]$ and $D[u]$, then they all write $D[D[u]]$. The result is the rooted tree shown in Figure 3.b.

The XMT platform 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 array D.

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: use two arrays to store the pointer graph, e.g. *D_read* and *D_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: $D_write[D_read[i]] = D_read[i]$. Note that you need to ensure that the updated pointer graph is stored in the appropriate array at the end of each iteration.

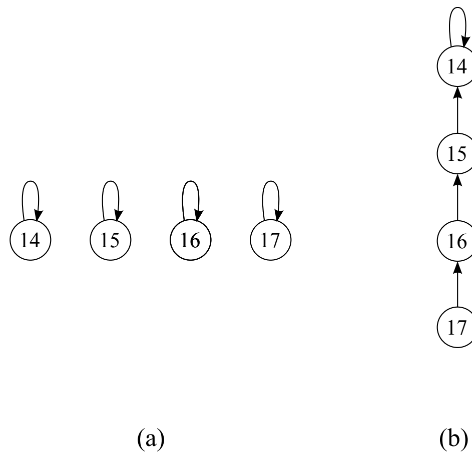

Figure 3: First iteration of the Second Connectivity algorithm applied on nodes 14-17 of the graph in Figure 2. (a) Initial state of the pointer graph. (b) The pointer graph after the *Hooking on smaller* step.

Resolving Concurrent Writes: In the second Connectivity algorithm, a rooted star is allowed to perform at most one hooking per iteration (either in step 2 or in step 3 of the algorithm). In the PRAM algorithm, the case where more than one hooking is possible is solved using concurrent writes in the Arbitrary CRCW convention.

For XMT, we recommend using prefix-sums operation combined with a *gatekeeper* array; for example, before hooking a rooted star onto a rooted tree using edge (u, v), execute a `psm()` instruction on an array entry corresponding to $D[u]$. Proceed with the hooking operation only if the prefix-sum returns 0.

5 Compiling and executing via the Makefile system

You can use the provided makefile system to compile and run your programs. For the smallest data set (graph0) parallel connectivity program is run as follows:

```
> make run INPUT=connectivity.p.c DATA=graph0
```

This command will compile and run the connectivity.p.c program with the dms1 data set. For other programs and data sets, change the name of the input file and the data set. You can use the `make check` command to compile and run your program and check the result for correctness.

```
> make check INPUT=connectivity.p.c DATA=graph0
```

If you need to just compile the input file (no run):

```
> make compile INPUT=connectivity.p.c DATA=graph0
```

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.

6 Grading Criteria and Submission

In this assignment, you will be graded on the correctness of your programs (connectivity.s.c and connectivity.p.c) and the performance of your parallel implementation in terms of completion time. Performance of your program will be compared against the performance of our reference implementation on graph2 and graph3 data sets. The cycle count for the reference implementation on graph2 is 1.3M cycles and on graph3 it is 1M cycles. For graph2 this corresponds to a speedup factor of 4 over our serial implementation of the connectivity algorithm. For graph3 the speedup factor is 7.6.

The correctness of your code will be checked via the `make check` rule as mentioned in the previous section. You can inspect the code in the `checks.c` file to see what is being checked. You have to store your results in the D, spanforest and spanforest_size global variables (see Section 4.2) in order for the checks to be performed correctly.

Please remove any printf statements that you may have placed for debugging purposes from your code as they will affect the performance and possibly break the automated grading script. Once you have the two source files ready you can check the correctness of your programs[1]:

```
> make testall
```

check the validity of your submission:

```
> make submitcheck
```

and submit:

```
> make submit
```

[1] `make testall` command runs all possible combinations of programs and data files. Since it blocks the FPGA for a long time, use this command judiciously in order not to inconvenience other users.