HW7: Graph Connectivity and Spanning Forests

Course:	Informal Parallel Programming Course for High School Students, Fall 2007
Title:	Graph Connectivity and Spanning Forests
Date Assigned:	November 20, 2007
Date Due:	December 11, 2007 *

* Submissions from students who are interested, but cannot make it by December 11, are also strongly encouraged. Such students are asked to send Scott Watson (*swatson@umd.edu*) and George Caragea (*george@cs.umd.edu*) an e-mail proposing a later date by which they intend to submit.

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 [1].

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 [1]. To derive a (not necessarily minimal) spanning forrest, 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 (similar to the BFS assignment). See Figure 1.

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 algoritms 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

The header files and the binary files can be downloaded from ∼ *swatson/xmtdata*. To get the data files, log in to your account in the class server and copy the *connectivity.tgz* file from directory using the following commands:

```
$ cp ~swatson/xmtdata/connectivity.tgz ~/
$ tar xzvf connectivity.tgz
```

This will create the directory *connectivity* with following folders: *data, src, doc*, and *results*. Data files are available in data directory. Put your *c* files to *src*, output file file to *doc* and result file to *results*.

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];
```

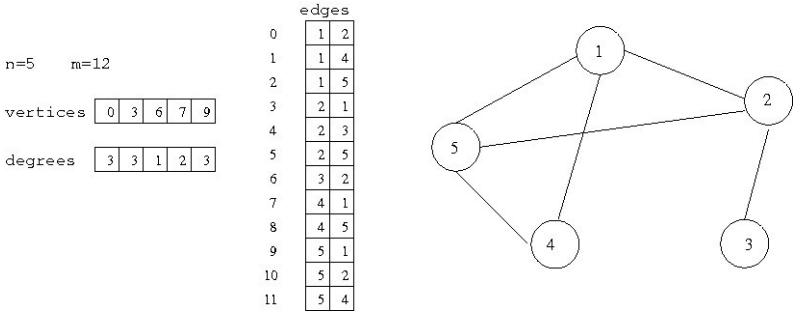

Figure 1: Incidence lists representation

```
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.32b
graph1	50	400	data/graph1/connectivity.h	data/graph1/connectivity.32b
graph2	10000	40000	data/graph2/connectivity.h	data/graph2/connectivity.32b
graph3	20000	40000	data/graph3/connectivity.h	data/graph3/connectivity.32b

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.

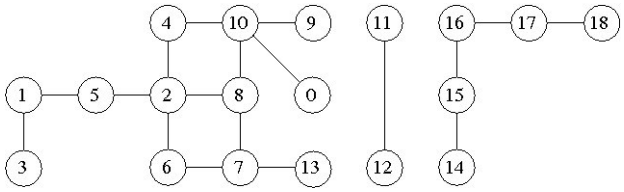

Figure 2: Graph example: `graph0` dataset.

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.

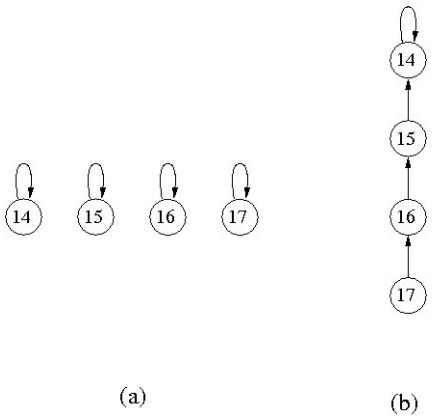

Figure 3: First iteration of the Second Connectivity algorithm applied on nodes 14-17 of the graph in Figure 2. (a) Initial state of the pointer graph. (b) The pointer graph after the *Hooking on smaller* step.

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, similar to the BFS problem; 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 Output

5.1 Cycle counts

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:	graph0	graph1	graph2	graph3
Parallel Clock cycles				
Serial Clock cycles				
Speedup				

Speedup is defined as $SerialClockCycles/ParallelClockCycles$.

5.2 Result

Following a run of your algorithm on the graph3 dataset, give the final pointer graph in the provided array $D[N]$ and the resulting spanning forest in $spanforest[N-1][2]$. Give the number of edges in the

spanning forest in the variable *spanforest_size*. Note that both the pointer graph and the spanning forest are not uniquely determined, and can even change between executions.

To collect this information, submit your implementation to the FPGA using the following command: `xmtfpga connectivity.b -d ../data/graph3/graph.32b -a 1004304 -l 240008`.

Retrieve the memory dump created by running the above job using the xmtfpgadb command. Place this file into the *results subdirectory* and rename it g3.txt. This file will be submitted and tested for correctness.

6 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, doc* and *results* subdirectories) using the make submitcheck command. Run following commands to submit the assignment:

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

If you have any questions, please send an e-mail to Scott Watson, *swatson@umd.edu*

References

[1] U. Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. In use as class notes since 1993. http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pdf, September 2007.