HW6: Breadth-First Search

Course:	Informal Parallel Programming Course for High School Students, Fall 2007
Title:	Breadth-First Search
Date Assigned:	November 13, 2007
Date Due:	November 27, 2007

1 Problem Statement

Breadth first search in parallel: Given a connected undirected graph $G(V,E)$ and a vertex $s \in V$, the breadth-first search (BFS) method visits vertices in the following order: First, visit s, then visit (in some order) all the vertices $w \in V$, where the edge $(s,w) \in E$; denote the set of these vertices by V_1, and the singleton set consisting of s by V_0; in general, V_i is the subset of vertices of V, which are adjacent on a vertex in V_{i-1} and have not been visited before (i.e., they are not in any of the sets $V_0, V_1, \ldots, V_{i-1}$). Each set V_i is called a layer of G and let h denote the number of layers in G.

The input graph will be stored using **incidence lists**. Let $V = 1, \ldots, n$ and $|E| = m$. In this representation, the edges are stored in an array of length $2m$. This vector will contain first all the edges incident on vertex 1, then all the edges incident on vertex 2, and so on. Note that each edge will appear twice in this vector (for an undirected graph).

This data structure will be provided using the following arrays:

- `edges[2m][2]`: the start and end vertex for each edge, grouped by the starting vertex.
- `vertices[n]`: the index in the `edges[]` array where the adjacent edges for each vertex begin.
- `degrees[n]`: the degree of each vertex
- `anti-parallel[2m]`: for each edge, stores the index in the `edges[]` array where its anti-parallel edge is stored.

An example is shown in figure 1.

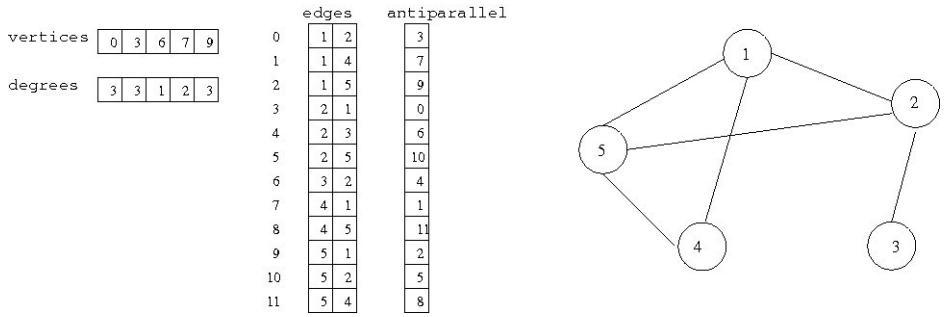

Figure 1: Incidence lists representation

2 Assignment

You will be required to submit three implementations with different solutions for the BFS problem:

1. Serial Implementation:
   - Describe a serial algorithm in the file algorithm.s.txt
   - Provide a brief work and time complexity analysis of this algorithm. Append this analysis to the file algorithm.s.txt
   - Write an XMTC program (XMTSER) that executes this algorithm. Name your code file bfs.s.c **Important:** The program should have as a result the lengths of the shortest path from the start vertex to all the other vertices stored in the array called `level`.
   - Run this program using the data sets given in the Input section.
   - Collect the number of clock cycles for each run into file table.txt (see Output section).
2. Nested Parallel Implementation:
   - Describe a parallel algorithm using nested spawns in the file algorithm.np.txt.
   - Provide a brief work and time complexity analysis of this algorithm. Do this analysis in two different ways: first taking into account that the inner spawn is serialized, and second assuming that the nested spawn is truly supported (all threads of the nested spawns run in parallel and are created in O(1) time). Append this analysis to the file algorithm.np.txt
   - Write an XMTC program (XMTNEST) that executes this algorithm. Name your code file bfs.np.c **Important:** The program should have as a result the lengths of the shortest path from the start vertex to all the other vertices stored in the array called `level`.
   - Run this program using the data sets given in the Input section.
   - Collect the number of clock cycles for each run into file table.txt (see Output section).
3. Parallel Implementation:
   - Describe a parallel algorithm in the file algorithm.p.txt. You are not allowed to used nested spawns in this algorithm.
   - Provide a brief work and time complexity analysis of this algorithm. Append this analysis to the file algorithm.p.txt
   - Write an XMTC program (XMTPAR) that executes this algorithm. Name your code file bfs.p.c **Important:** The program should have as a result the lengths of the shortest path from the start vertex to all the other vertices stored in the array called `level`.
   - Run this program using the data sets given in the Input section.
   - Collect the number of clock cycles for each run into file table.txt (see Output section).

3 Input

3.1 Setting up the environment

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

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

This will create the directory *bfs* with following folders: *data, src*, and *doc*. Data files are available in data directory. Put your *c* files to *src*, and *txt* files to *doc*.

3.2 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)
edges[M][2]	The start and end vertex of each edge
vertices[N]	The index in the `edges` array, at which point the edges incident to vertex begin
degrees[N]	the degree of each vertex
antiparallel[M]	for each edge, stores the index in the `edges` array where its anti-parallel edge is stored.
gatekeeper[N]	gatekeeper per vertex
locks[M]	Array of locks
level[N]	result array: stores the distance from the source for each vertex

In order to use the same code and dataset with multiple starting nodes, you will assume that a C preprocesor variable `START` is available. During compilation you will be able to modify this preprocesor variable using `-D START=...` compiler option. This has the same effect as if the first line of your code has the compiler directive: `#define START ...` (substitute an integer in place of `...`). for grading purposes, your program will be compiled and run with different starting nodes using this method. If you do not adhere to this convention, your assignment may not be graded fully.

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() {
 //...
}
```

3.3 Data Sets

The following two datasets are provided:

Data set	n = # Vertices	m = # Edges	Start node	Header file	Binary File
Hexagon	20	86	0	data/hexagon/bfs.h	data/hexagon/bfs.32b
Large	1000	10000	142	data/large/bfs.h	data/large/bfs.32b
Huge	10000	100000	101	data/huge/bfs.h	data/huge/bfs.32b

The below list provides details for each data set.

1. **Hexagon graph:** The dataset `hexagon` corresponds to the graph in figure 2.
2. **Large:** This is a rather large graph, generated with an automated tool. In order to test for correctness of your algorithm, you can check that the following (node:level) pairings hold: (199:2), (300:3), (900:3), (401:3).
3. **Huge:** This is an even larger graph, generated with an automated tool.

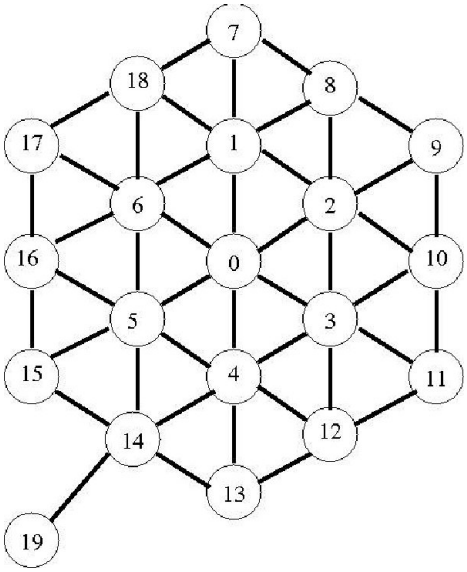

Figure 2: The `hexagon` dataset

4 Output

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:	Hexagon	Large	Huge
XMTPAR Clock cycles			
XMTNEST Clock cycles			
XMTSER Clock cycles			

5 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
```

6 Hints and remarks

Note that an efficient parallel solution might require nested spawning. The current XMTC compiler implementation allows nesting `spawn` statements but serializes the inner spawns, so it does not support nested *parallelism* through nesting spawn statements. You are asked to start by writing a parallel algorithm using nested spawns because it is easier think in terms of nested spawns than single-spawns (see below) and a program with nested spawns can be transformed into one that only uses single-spawns (within the outer spawn) relatively painlessly. Additionally, once nested spawns are fully supported, this is how programmers will be expected to write code.

For the third algorithm where you are not allowed to use nested spawns, you can use another construct called a single-spawn (`sspawn()`). A running thread can start exactly one more thread by using the

`sspawn()` (*single-spawn*) instruction. By using a balanced binary tree type approach, a thread can start n more threads in $\log n$ steps. For details about the `sspawn()` instruction, you are referred to the **XMT manual** and the **Appendix A** of this document.

In implementing the parallel solution, you might come across a case where two or more threads try to write to the same memory location simultaneously. The XMT platform forbids arbitrary concurent writes, the only mechanism that can be used in this case being prefix-sum to memory `psm()`.

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

A Clarifications on using single-spawn

When using single-spawn to start a new thread, a synchronization step between the parent and the child thread is necessary. Since the child thread can potentially start executing as soon as a new thread-id is allocated, the synchronization will ensure that the initialization block for the child is executed (by the parent) before the child starts.

In the current version of the XMT framework, it is the responsibility of the programmer to implement this synchronization, using a busy-wait technique. For this purpose, a special global array will be used; the child thread will keep reading from a certain location in this array and will not proceed until the parent writes a value in that location, signaling that the initialization data is in place. To ensure atomicity of reads and writes, prefix-sum to memory operations will be used to read and write the shared lock variables.
Example:

```
int locks[100];
...
spawn(low,high) {
  int child_ID;
  int lock;
  if (thread is a single-spawned thread) {
    lock = 0;
    while (lock==0) {        // spin wait
      psm(lock,locks[$]);
    }
  }
  ...
  sspawn(child_ID)
  {
    ... Initialization Block: Code for newly spawned thread
    lock=1;
    psm(lock,locks[child_ID]); // give signal to child
  }
  ... Some other code here ...
}
```

All the other guidelines and restrictions regarding the usage of `sspawn` described in the XMTC Manual and the XMTC Tuturial still apply.