HW 6B: Fine-Tuning Breadth-First Search

Course:	ENEE759K/CMSC751, Spring 2010
Title:	Fine-Tuning Breadth-First Search
Date Assigned:	April 22, 2010
Date Due:	May 11, 2010 11:59pm
Contact:	George Caragea - george@cs.umd.edu

1 Assignment Goal

The goal of this assignment is to improve upon your implementation of BFS from HW4. You will fine-tune your implementation to provide the best results on graphs with different properties.

The nature of the computation in BFS is highly dependent on the characteristics of the input graph. For this assignment, we provide you with graphs that have very different characterstics, for which none of the implementations you provided in HW4 should perform optimally. Instead of providing you with guidelines on the precise algorithm to use, you are to experiment with different implementations, and provide one solution that works best on all the datasets provided.

2 Assignment

Implement an improved parallel algorithm for the BFS problem described in HW4: BFS. Name your code bfs.opt.c.

3 Input

3.1 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 *bfstuning.tgz* using the following command:

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

This will create the directory *bfstuning* which contains the C file templates that you are supposed to edit 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/bfs*. 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.

3.2 Input Format

The type and size of the data structures provided is given in the following table. Note that the format is identical to the one posted for HW4: BFS.

#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

Attention: You don't have to use all the auxiliary arrays defined in the header file for an optimal solution. They are provided here for your convenience, but you can declare your own global arrays within your program.

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 `-DSTART=...` 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.

3.3 Data Sets

The following three datasets are provided:

Data set	$n =$ # Vertices	$m =$ # Edges	Start node	Header file	Binary File
huge	10000	200000	101	$DATA/huge/bfs.h	$DATA/huge/bfs.xbo
star	50001	100000	0	$DATA/star/bfs.h	$DATA/star/bfs.xbo
web	50000	278934	0	$DATA/web/bfs.h	$DATA/web/bfs.xbo

The below list provides details for each data set.

1. **huge:** This is a random graph. Edges are added at random with uniform probability between any two nodes. Note that this is the same dataset provided as part of HW4: BFS.
2. **star:** This is a star-shaped graph. Node 0 is the center of the star, and all the other nodes are connected to it, and only to it.
3. **web:** This is a graph representing a small subsection of the web. with edges representing hypelinks between web pages. The degrees of the nodes follow a power law distribution: a few nodes represent very popular pages and have very high degrees (5000+), while most of the nodes have fairly low degrees.

$DATA is */opt/xmt/class10/xmtdata/bfs*. Note that each edge is listed twice in the input file. For example, the undirected graph Huge has 100,000 edges, but it contains 200,000 directed edges in the adjacency list, listed in the array `edges`

4 Compiling and executing via the Makefile system

You can use the provided makefile system to compile and run your programs. For the star data set, the program is run as follows:

```
> make run INPUT=bfs.opt.c DATA=star START=0
```

This command will compile and run the bfs.opt.c program with the star data set and a starting node 0. 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=bfs.opt.c DATA=star START=0
```

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.

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 using the make submitcheck command. Run following commands to submit the assignment:

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

6 Reference Performance Results

The cycle counts for our reference implementation for some of the input datasets are in Table 1.

Program	star	web
bfs.opt.c	1,192,123	3,197,729

Table 1: Reference cycle counts

7 Grading Criteria

In this assignment, you will be graded on the correctness of your program 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 all input graphs.

8 Hints and Remarks

You might want to start by running your solution from HW4: BFS using the new datasets and examine the cycle counts. If you followed the guidelines there, you should see that for some datasets, the nested implementation will run faster, while for others, the fully parallel one will.

The reason why the fully parallel is slower on some datasets (even though it provides more parallelism) is that starting threads through the single-spawn mechanism can be pretty expensive. Moreover, if all the parallel cores are busy, there is no benefit in starting more threads. In cases where only a few edges are to be processed, it might be beneficial to process them serially instead of starting additional threads. However, if the number of edges to process is large, ramping up more threads to process them is better. You should experiment with combining the two approaches and finding the right parameters for an implementation that works best (or close enough) for all the input datasets provided.