Lab 4: Breadth-First Search

In this lab, you will implement a bag data structure and use this bag data structure to implement a parallel breadth-first search. The write-up for this lab is due on Wednesday, March 17 at 11 am.

Reading

- Section 22.2 of CLRS [1].
- Charles E. Leiserson and Tao B. Schardl, "A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers)," Submitted for publication, 2010. Web link provided on Stellar.

1 Serial Breadth-First Search

Breadth-first search is a simple but important algorithm for searching a graph. Given a graph $G = (V,E)$ with a ***source*** vertex s, a breadth-first search (or ***BFS*** for short) from s discovers the vertices that are reachable for s, and computes the distance (in number of edges) from s to these vertices. A breadth-first search discovers all vertices at distance k from s before discovering any vertices at distance $k+1$.

One standard approach to breadth-first search (e.g. in CLRS, Section 22.2) is to colors vertices as they are visited. Initially every vertex u starts as white, but u changes to gray when it is first discovered. When the breadth-first search has finished discovering all vertices v adjacent to u, then u changes to black. The BFS algorithm in [1] uses a first-in-first-out (FIFO) queue Q to manage the set of gray vertices, as shown in Figure 1.

(a) For this lab, we have provided code for a simple breadth-first search using a FIFO queue. You can check out code using the following command:

`git clone /afs/csail.mit.edu/proj/courses/6.884/spring10/labs/lab4/ lab4`

The provided breadth-first search code runs on test graphs which are stored in binary format. Since these test graphs are quite large, we have not checked them into the code repository. Instead, we have copied the test files into the `/scratch/6.884` directory on each of `cagnode`.

Alternatively, you can copy the test files from the directory:

`/afs/csail.mit.edu/proj/courses/6.884/spring10/labs/lab4_data`

The input files are compressed using `bzip2`; nevertheless copying these files may take quite a while. You may also run out of disk quota for your CSAIL AFS account if you attempt to uncompress all of the the files into your home folder. In general, on each `cagnode`, you can create a temporary directory for yourself in `/scratch/` to store large files. Note that `/scratch` is a drive local to the particular node you are logged into, so any files you create in `/scratch` are not visible if you log into a different node.

To build the main program and run a serial breadth-first search on a test graph, type the following commands:

```
BFS(G,s)
 1  for each vertex u ∈ V[G] − {s}
 2       color[u] = WHITE
 3       d[u] = ∞
 4       π[u] = NIL
 5  color[s] = GRAY
 6  d[s] = 0
 7  π[s] = NIL
 8  Q = ∅
 9  ENQUEUE(Q,s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ Adj[u]
13            if color[v] = WHITE
14                 color[v] = GRAY
15                 d[v] = d[u] + 1
16                 π[v] = u
17                 ENQUEUE(Q,v)
18       color[u] = BLACK
```

Figure 1: Pseudocode for a serial breadth-first search, from CLRS, Chapter 22 [1]. In this code, for every vertex u, $d[u]$ stores the distance of u from s. The vertex $\pi[u]$ stores the predecessor of u, that is, the BFS first discovers u by following the edge $(\pi[u], u)$.

```
cagnode1:~$: make bfs_driver
cagnode1:~$: ./bfs_driver /scratch/6.884/kkt_power.bin 1 --check
```

This command executes a BFS of type 1 on the input graph in the file `kkt_power.bin`, checks the computed distances, and generates output messages to the console. Type 1 executes the method `bfs_serial`, which is a serial version of BFS similar to the one shown in Figure 1. An `enum` in `bfs_driver.cilk` describes the possible types of BFS that can be executed. In later parts of this lab, you will complete the implementations of these other versions of BFS. The `bfs_driver` file also documents some of the other command-line arguments that `bfs_driver` supports.

The driver program reads the input graphs in from file, and stores the graphs in a compressed-sparse-row (CSR) format. The driver program executes a BFS starting from node 0 of the graph. For more details on the graph implementation, see `graph.h` and `graph.cilk`.

Unfortunately, in our initial implementation of breadth-first search, the FIFO queue represents a serial bottleneck, since the queue imposes an ordering constraint, that vertices are processed one at a time, in the order they are first discovered. It turns out, however, that one can relax this constraint and still compute the correct distances from the source. Define ***layer*** k, denoted L_k, as the set of all vertices which are at a distance

exactly k from the source s. Intuitively, a breadth-first search algorithm is free to process the vertices within a given layer L_k in any arbitrary order, as long as each of the layers is processed sequentially, that is, for all k, layer L_{k-1} is processed before layer L_k.

(b) Argue that the queue in the breadth-first search algorithm in Figure 1 never contains nodes from more than two distinct layers.

Create a serial version of breadth-first search that uses two separate queues, one for each layer. A prototype for this version of BFS has already been provided for you (the `bfs_2queue` function in `graph.cilk`).

2 Using Bags for BFS

To parallelize breadth-first search, we would like to be able to traverse one layer and generate the next layer in parallel. Instead of storing each layer in a queue, we will the use a ***bag*** data structure to maintains an unordered set. The bag supports operations that allow one to:

- ***create*** an empty bag,
- ***insert*** an item into a bag,
- ***union*** the contents of two bags, destroying one bag and modifying the other, (in the code this operation is labeled "merge," since "union" is a reserved keyword in C/C++.)
- ***split*** a bag into two disjoint bags (with some fraction of the items moving to a new bag, and the rest of the items remaining in the current bag), and
- ***walk*** the bag data structure, visiting all the items.

For breadth-first search, we can maintain two bags, one for layer L_k and one for L_{k+1}, and then walk through the bag for L_k to generate the nodes to insert into the bag for L_{k+1}.

To parallelize the BFS, we can adopt a divide-and-conquer approach; split an input bag for L_k into several smaller input bags, walk each of the small input bags in parallel and generate a small output bag for layer L_{k+1}, and then union the small output bags together into one final output bag for layer L_{k+1}. In this lab, we use a Cilk++ reducer hyperobject for maintaining bags; the reduce function unions two output bags together.

One straightforward implementation for a bag is to maintain elements as a linked list. With a linked list, a union operation on two linked lists requires only constant time. Conceptually, it is also easy to split a linked list into pieces if the split point is given. Unfortunately, efficiently finding a split point which creates two pieces of roughly equal size seems difficult. Nevertheless for this lab, we provide a linked-list implementation of a bag, since this implementation illustrates the bag interface and may be useful for debugging your parallel breadth-first search implementation in later parts of the lab.

(c) We have provided a generic interface for the bag data structure and the associated reducer in `bag.h`. We have also provided an implementation of the bag using a linked list in `bag_list.cpp`, and test cases for the bag interface in `bag_test.cpp`. Compile and verify that the provided linked-list implementation of a bag passes the cases.

```
cagnode1:~/lab4$: make bag_test; ./bag_test
```

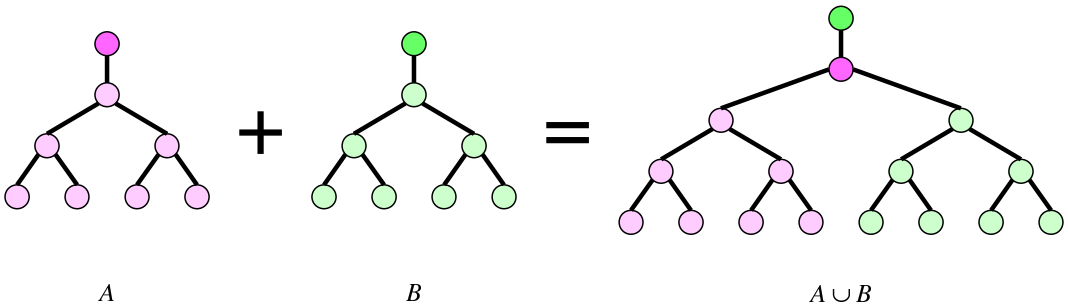

Figure 2: Two pennants of equal size can be unioned in constant time.

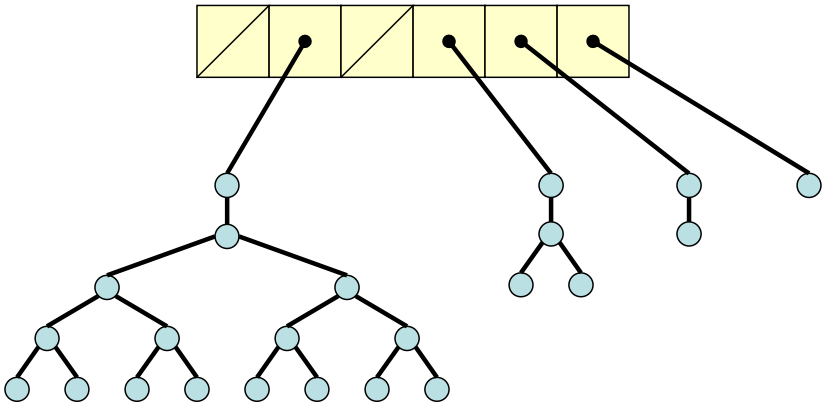

Figure 3: A bag with 23 elements.

(d) Implement a serial BFS which uses the linked-list implementation of bags instead of queues to store the nodes in a layer. The prototype of this BFS version has been provided (`bfs_bag_list` in `graph.cilk`). The linked-list bag implementation provides a `split` method which splits off a constant number of elements from the front the list for the original bag, and returns a new bag containing these elements. The subroutines `bfs_walk_layer` and `bfs_walk_layer_base` may also be useful for your implementation.

(e) Modify your BFS implementation so the pieces of a bag are traversed in parallel. This parallel BFS actually contains a data race; is this race a bug? How does your implementation perform as compared to the serial version? Why might we expect this implementation to have limited scalability?

3 A Parallel Bag Data Structure

For parallel BFS, we want to improve the linked-list implementation of a bag to provide a more efficient `split` method, that is, a split method that creates bags of roughly equal size. This improved bag data structure is defined in terms of an auxiliary data structure, called a "pennant."

A ***pennant*** is a tree where the root has only one child consisting of a complete binary tree on 2^k elements for some integer k. As illustrated in Figure 2, two pennants A and B both of size 2^k may be unioned to form

a pennant of size 2^{k+1} by the following steps.

1. Modify the root of A by adding a second child; the child of B's root becomes the second child of A's root.
2. Modify the root of B so that its only child is the root of A.
3. Return the root of B as the root of the new unioned pennant.

A bag is a collection of pennants, each of a different size. A bag may be represented by an array, list, or other data structure with pointers to the pennants it contains. One implementation of a bag uses an array where the kth component of the array contains either a null pointer or a pointer to a pennant of size 2^k. We shall use this representation for descriptive purposes. A pennant x_k of size 2^k may be added to a bag S as follows:

1. If $S[k] = \text{NULL}$, set $S[k] = x_k$ and terminate.
2. If $S[k] = y_k$, where y_k is a pennant of size 2^k, union x_k and y_k to form a pennant x_{k+1} of size 2^{k+1}, set $S[k] = \text{NULL}$, and recursively add x_{k+1} to S.

Note the similarity of this process to that of incrementing a binary counter.

Given three pennants x, y, and z, where each either has size 2^k or is empty, we may union them to produce a pair of pennants $(s,c) = f(x,y,z)$, where s has size 2^k or is empty and c has size 2^{k+1} or is empty. The following table details the process by which f is computed, where 0 means that the pennant is empty and 1 means that it has size 2^k:

x	y	z	s	c
0	0	0	NULL	NULL
1	0	0	x	NULL
0	1	0	y	NULL
0	0	1	z	NULL
1	1	0	NULL	UNION(x,y)
1	0	1	NULL	UNION(x,z)
0	1	1	NULL	UNION(y,z)
1	1	1	x	UNION(y,z)

The following pseudocode uses this process to union two bags A and B using an auxiliary variable x to hold a pennant:

1. $x = \text{NULL}$
2. For $k = 0$ to n do
3. $\quad (A[k],x) = f(x,A[k],B[k])$

(f) Implement the pennant data structure, and use pennants to create a new bag implementation. For your convenience, we have provided an interface for your bag data structure in `bag_parallel.cpp`.

(g) Use your improved bag implementation to implement a parallel breadth-first search. In `graph.cilk`, the prototype of your method, `bfs_parallel`, has been provided.

If your bag implementation satisfies the interface in `bag.h`, (as `bag_parallel.cpp` does), you should not need to define your own class for the Cilk++ reducer hyperobject. In `bag.h`, `Bag_reducer` is a class template which given any bag implementation implementing the methods for `Bag` class template, wraps this implementation for use as a Cilk++ reducer hyperobject.

How does the absolute performance of your parallel BFS compare to performance of the original serial BFS on one processor? Does your parallel BFS exhibit parallel speedup?

(h) A straightforward implementation of pennants and bags may have high space overhead and be expensive to traverse if the leaves of each pennant contain only a single element. Coarsen the base cases of your bag and/or parallel BFS implementations to improve performance. For convenience, we have provided an interface in `bag_opt.cpp`, as well as a method `bfs_parallel_opt` method that you can complete. Use Cilkview to verify that parallel BFS implementation is exposing significant parallelism.

After optimizing your bag and BFS implementation, you may discover that despite the high parallelism reported by Cilkview, you are not achieving significant parallel speedups. For a BFS which performs relatively little work at each node, performance may be limited due to insufficient memory bandwidth. A BFS that performs more work at each node may exhibit better scalability because it has greater a ***arithmetic intensity***, i.e., number of arithmetic operations performed on a given memory location.

(i) To observe the effect of changing arithmetic intensity, implement a parallel BFS which artificially increases the work done at each node. We have provided a prototype for this BFS (`bfs_memtest`), which takes in an extra parameter for the amount of artificial work to add. To create artifical work, we have provided an external library function, `trivial()`, which generates empty function calls. (This function is provided as an external library to prevent it from being optimized away by the compiler.) Test how your parallel BFS scales as you vary the amount of work performed at each node.

4 Further Exploration

The parallel breadth-first search you have implemented in this lab, using reducers and the bag data structure, is modeled after the algorithm described by Leiserson and Schardl in [2]. In their paper, they also analyze the running time of a parallel breadth-first search algorithm similar to the one described in this lab, but where locks are used to eliminate benign data race. More precisely, for a graph $G = (V,E)$ with diameter D, their algorithm runs in expected time $O((V+E)/P + D\lg^3(V/D))$ on P processors.

After implementing your parallel BFS, see if you can extend your work in some interesting way. Some ideas for possible extensions include:

- Using PBFS to compute the transitive closure of a sparse graph.
- Implementing an efficient iterator for your bag data structure. What is the worst-case cost of moving your iterator to the next element? What is the amortized cost of using the iterator to walk the entire bag?
- Investigating whether preprocessing your graph and changing the layout can improve performance when you need to run BFS multiple times on the same graph (possibly from different source vertices).
- Analyzing the runtime of parallel breadth-first search when you allow for benign data races. As we mentioned in Section 3, the parallel BFS implementation we present actually contains a benign data race, which can theoretically increase the work of the computation. To avoid this problem, the analysis in [2] requires additional synchronization using locks; in practice, however, using locks to eliminate the race may hurt performance for many graphs. Are there ways to keep some benign data races in practice while still guaranteeing good theoretical properties in the worst case?

Be wary of unconsciously turning this assignment into your term project, however. You should spend only about 12 hours on this lab.

If you are interested in extending this lab into a term project, some ideas include:

- Parallelizing an application which uses breadth-first search (e.g., a model checker).
- Using parallel breadth-first search to compute maximum flows.
- Investigating whether parallel BFS can be extended to work for computing shortest paths. (This idea may be hard.)
- Parallelizing other graph algorithms.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[2] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). Submitted for publication, 2010.