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 color 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 \( u \in V[G] - \{s\} \)
2. \( \text{color}[u] = \text{WHITE} \)
3. \( d[u] = \infty \)
4. \( \pi[u] = \text{NIL} \)
5. \( \text{color}[s] = \text{GRAY} \)
6. \( d[s] = 0 \)
7. \( \pi[s] = \text{NIL} \)
8. \( Q = \emptyset \)
9.\( \text{ENQUEUE}(Q, s) \)
10. while \( Q \neq \emptyset \)
11. \( u = \text{DEQUEUE}(Q) \)
12. for each \( v \in \text{Adj}[u] \)
13. if \( \text{color}[v] = \text{WHITE} \)
14. \( \text{color}[v] = \text{GRAY} \)
15. \( d[v] = d[u] + 1 \)
16. \( \pi[v] = u \)
17. \( \text{ENQUEUE}(Q, v) \)
18. \( \text{color}[u] = \text{BLACK} \)

**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 \( (\pi[u], u) \).

```bash
$ 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 \( \text{layer} k \), 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 use a bag data structure to maintain 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.

```
cagnodel:˜/lab4$ make bag_test; ./bag_test
```
(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 $k$th 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 $2^k$:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
<th>$s$</th>
<th>$c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>NULL</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>y</td>
<td>NULL</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>z</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>NULL</td>
<td>UNION($x,y$)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>NULL</td>
<td>UNION($x,z$)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>NULL</td>
<td>UNION($y,z$)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$x$</td>
<td>UNION($y,z$)</td>
</tr>
</tbody>
</table>

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. $(A[k],x) = f(x,A[k],B[k])$

(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?
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.

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 artificial 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 $O((V + E)/P + D \log^3(V/D))$ 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 Optimizing Bag Implementation

1.1 Description

My optimized bag implementation employs three optimizations. First, the pennants in the bag hold multiple values. Secondly, to make this optimization efficient, I enforce an invariant that each pennant must be full. Since this is not always possible, I maintain a buffer pennant – which is a pennant with only one node that can be less than full. Then, when this buffer pennant is full, I merge it with the rest of the pennants. This complicates merging bags a little, but does not affect the theoretical runtime or greatly affect the actual runtime.

Additionally, in my BFS walk of a bag, rather than splitting a bag all the way down to a base case of \( n = 1 \), where \( n \) is the number of nodes in the bag, I set a coarsening cutoff. After the BFS has split some number of times (which should be a factor of \( \log \) of the original bag size) I stop splitting.

1.1.1 A Note about Tests

I excluded path.bin from my tests. It performs very poorly for all my bag implementations. I think there are two reasons for this. First, initializing 6 million reducers adds a lot of overhead here. Second, my code contains a few memory leaks and once that many reducers become instantiated, my code almost collapses. I’m new to C++ so in the future I’ll be more careful about pointers.

1.2 Optimization Parameters

There are a couple parameters for the optimized bag, namely the size of each pennant and the factor for the BFS cutoff (I call this depth.) I found the best values for these by testing various parameters on the different input sets. The optimal penant size I found for my implementation is 64 elements. The optimal value for depth was \( 0.2 \log_{10} n \) splits. For most inputs this results in 4 splits.

1.3 Results

1.3.1 Parallelism

Before tuning, this algorithm should exhibit a large amount of parallelism on the different input graphs. Here, because of the coarsening, I expect parallelism to be not too much higher than 8. I ran cilkview on the optimized bag algorithm with the rmat input set.

1) Parallelism Profile
   Work : 2304795289 instructions
   Span : 91071042 instructions
   Burdened span : 91071042 instructions
   Parallelism : 25.31
Burdened parallelism : 25.31
Number of spawns/syncs: 8388855
Average instructions / strand : 91
Strands along span : 63
Average instructions / strand on span : 1445572
Total number of atomic instructions : 18332
Frame count : 16777714

2) Speedup Estimate
2 procs: 1.87 - 2.00
4 procs: 3.33 - 4.00
8 procs: 5.44 - 8.00
16 procs: 7.97 - 16.00
32 procs: 10.38 - 25.31

From the cilkview output I concluded that this solution exhibited enough parallelism. However, I am not experiencing the speedups that this parallelism suggests I may be able to achieve. This is likely due to memory bandwidth problems, which I examine in a later section.

1.3.2 Timing

![Runtime comparison of BFS using parallel optimized bags and a normal serial BFS. Run-times are averaged across 3 executions.](image)

As Figure 1 illustrates, the parallelized BFS using optimized bags runs significantly faster than the serial BFS.
2 Memory Bandwidth Results

In this lab I experimented with how BFS pushed the memory bandwidth by adding trivial operations to each node in the graph. This artificially increases the workload of the BFS search, which may cause the algorithm to scale more efficiently, since the algorithm will no longer be memory bounded.

![Graph showing speedup](image)

Figure 2: This figure shows the speedup up `bfs_driver` running on rmat32 with 3 different amounts of trivial operations (5000, 1000, 100.) As you can see, at 5000 trivial operations per node, the algorithm exhibits linear speedup.

3 Alternative Split Method

Additionally, I explored an alternative bag split method which takes constant time to perform a split, while sacrificing some of the balance from the typical bag split. In this section I refer to the normal split method described in the paper *A Work-Efficient Parallel BFS Algorithm* as balanced and my alternative split method as unbalanced.

3.1 Description / Theoretical Analysis

In the balanced bag split, the bag is split almost perfectly in two by iterating through each pennant and splitting each pennant. A simpler approach may be to just split the highest pennant in a bag or if a pennant exists one step lower, just removing that instead (see code in the next subsection.) This way, we guarantee that the bags are at worst a 3 to 1 ratio, while only taking \(O(1)\) time. It does not alter theoretical parallelism, because the span still reaches the base case for a bag in \(O(\log n)\) splits, where \(n\) is the original size of the bag. I implemented this strategy to see if it performs better in practice.
3.1.1 Unbalanced Split Code

Bag<T, BAG_OPT>::simple_split(){
    int high = this->highest;
    if(high <= 1){
        return NULL;
    }
    Bag<T, BAG_OPT>* r = new Bag<T, BAG_OPT>();

    PenantOpt<T>* temp;

    if(this->S[high - 1] != NULL){
        temp = this->S[high - 1];
        this->S[high - 1] = NULL;
    }else{
        temp = (this->S[high]->split(high));
        this->S[high - 1] = this->S[high];
        this->S[high] = NULL;
        this->highest--;
    }

    r->S[high - 1] = temp;
    r->highest = high - 1;

    return r;
}
3.2 Performance

Figure 3: Runtime comparison of BFS using optimized bags with either balanced or unbalanced split operations. Runtimes were averaged across 3 executions.

From Figure 3 you can see that the unbalanced split performs well on most inputs, even outperforming on one input. This suggested to me that perhaps a hybrid approach would benefit performance.

3.3 Hybrid Performance

Figure 4: Runtime comparison of BFS using optimized bags with either balanced or hybrid split operations. Runtimes were averaged across 3 executions.
I implemented a hybrid of the two split methods by balanced splitting for large bags and unbalanced splitting for small bags. I controlled this by switching split methods after a certain number of splits occurred. While not optimal, it was a crude estimate for how a hybrid approach might perform. From 4 we find that the hybrid approach does not increase performance past that of balanced splitting in general, though it sped up over simply unbalanced splitting for most inputs and even beats balanced splitting in some inputs. This suggests that perhaps with more tuning, I would experience some speedup. However, since we know from performing memory tests that so much of the performance is bottlenecked by memory bandwidth, this may not be a particularly fruitful approach.
Lab 4 Writeup

Introduction

For this lab we created and compared the performance of several implementations of the breadth-first search algorithm. First, we looked at a basic serial implementation of BFS which uses a single FIFO queue to store nodes for each level to be examined. Next, we parallelized BFS by using bag data structures to enable traversing one layer while simultaneously creating the next layer. We tried two different ways to implement the bag data structure: linked lists and pennants. Next, we coarsened the pennant bag data structure implementation to improve constant time factors. We observed that while cilkview shows high parallelism for our coarsened parallel BFS, we didn’t achieve significant parallel speedups. We show that this is due to insufficient memory bandwidth. Finally, we discuss our iterator and a technique we used to speed up the pennant bag data structure.

For the purposes of this lab, our breadth-first search algorithms were run on graphs with integer node values, edge weights of 1, and diameters relatively small to the number of nodes and edges. The graphs we had access to are summarized in Table 1.

Table 1: Sizes of Test Graphs.

<table>
<thead>
<tr>
<th>Graph Type</th>
<th>#Nodes</th>
<th>#Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>layered</td>
<td>40000</td>
<td>39000000</td>
</tr>
<tr>
<td>grid3d – 200</td>
<td>800000</td>
<td>55760000</td>
</tr>
<tr>
<td>kkt_power</td>
<td>2063494</td>
<td>12771361</td>
</tr>
<tr>
<td>rmat23 – denser – unbalanced</td>
<td>8388608</td>
<td>78753383</td>
</tr>
<tr>
<td>vanHeukelum – cage15</td>
<td>5154859</td>
<td>99199551</td>
</tr>
<tr>
<td>vanHeukelum – cage14</td>
<td>1505785</td>
<td>27130349</td>
</tr>
<tr>
<td>path</td>
<td>2000000</td>
<td>19999999</td>
</tr>
</tbody>
</table>

Based on runtimes (see Table 2), node to edge ratio, diameter size, and what we thought most other groups would choose, we use only kkt_power to test our parallelized BFS algorithms.

1 Serial Breadth-First Search

Table 2: serial_BFS and serial_BFS2 runtimes.

<table>
<thead>
<tr>
<th>Graph Type</th>
<th>serial_BFS</th>
<th>serial_BFS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>layered</td>
<td>114 ms</td>
<td>114 ms</td>
</tr>
<tr>
<td>grid3d – 200</td>
<td>2699 ms</td>
<td>2672 ms</td>
</tr>
<tr>
<td>kkt_power</td>
<td>622 ms</td>
<td>627 ms</td>
</tr>
<tr>
<td>rmat23 – denser – unbalanced</td>
<td>1626 ms</td>
<td>1614 ms</td>
</tr>
<tr>
<td>vanHeukelum – cage15</td>
<td>1406 ms</td>
<td>1419 ms</td>
</tr>
<tr>
<td>vanHeukelum – cage14</td>
<td>337 ms</td>
<td>344 ms</td>
</tr>
<tr>
<td>path</td>
<td>430 ms</td>
<td>426 ms</td>
</tr>
</tbody>
</table>

First, we recorded the runtime of serial BFS (Table 2) to serve as a benchmark for the rest of this lab. In addition to the standard queue implementation of serial_BFS, we implemented a double queue BFS, serial_BFS2, with one queue for the current layer and another for the next. This serves as a more direct comparison to our bag BFS’s, which have one bag for the current layer and another for the next. We took the average of 5 trials for each of our graphs, where
each trial ran on an 8-core cagnode machine. As shown in Table 2, the runtimes for \textit{serial\_BFS} and \textit{serial\_BFS2} were nearly identical.

2 Parallelized Linked-List Bag Breadth-First Search

By implementing the BFS node queue as a bag with an underlying linked-list structure, we were able to parallelize the algorithm, but weren’t able to reduce the runtime. By taking the average of 5 trials on \textit{kkt\_power} on 8 cores with a cagnode machine, we found that parallelized bag list BFS ran in 750 ms (up from the 627 ms of the serial algorithm runtime).

For this implementation, each iteration of BFS splits the current bag into sub-pieces of size k, traverses each piece in parallel and then merges the resulting bags back into one. Not only does each split take $\Theta(k)$, but it must be done serially, which leads to burdened parallelism as shown in Figure 1, making this algorithm a poor choice for parallel BFS.

![Figure 1: Parallelized Linked-List Bag Breadth-First Search](image)

3 Uncoarsened Parallelized Pennant Bag Breadth-First Search

Next, we implemented the pennant data structure described in "A Work-Efficient Parallel Breadth-First Search Algorithm" by Charles E. Leiserson and Tao B. Schardl. We used it to write a parallelized pennant bag BFS, \textit{PBFS}.

Since the \textit{PBFS} algorithm visits each vertex and edge exactly once, the total work done by this algorithm is $O(V + E)$. For each layer, this algorithm splits the layer into pieces of a certain grainsize and traverses these pieces in parallel. Thus, for each layer the algorithm must (1) create temporary bags (2) split the layer into sub-pieces (3) insert the next layer’s values into the next layer data structure (a pennant bag reducer). As discussed in the paper by Charles E. Leiserson and Tao B. Schardl, this algorithm does $O(V + E)$ amount of work and has a span of $O(D \cdot lg(V/D)) + D \cdot lg(max degree(V))$ where D is the number of layers in the graph.

By taking the average of 5 trials on \textit{kkt\_power} on 8 cores with a cagnode machine, we found that parallelized bag list BFS ran in 9155 ms, which was much higher than the serial algorithm runtimes. Long runtimes were due to a lack of coarsening, rather than a lack of parallelism as shown in Figure 2.

Based on the results shown by cilkview and the obtained runtimes of this algorithm, we observed that this algorithm exhibited good parallelism but had poor runtime performance. In the next section we discuss how we mitigate this poor runtime performance by sacrificing some of the high parallelism to reduce overhead.
To improve the runtime performance of this algorithm we attempted to coarsen the base case in 2 different ways.

First, we coarsened the algorithm by having a minimum threshold bag size, $k$, for recursive bag splitting. If a bag contained fewer than $k$ pennants, we did not continue to recursively split the bag into sub-pieces. By doing this, we observed that the runtime significantly decreased, presumably by reducing the overhead of performing BAG-SPLIT on arbitrarily small bag sizes.

Again, we took the average of 5 trials on kkt_power on 8 cores with a cagnode machine. We found that $k = 13$ produced the fastest run time, which on average was 453 ms. $k = 13$ yielded speedups of 20 times with respect to $k = 2$ (uncoarsened) and ran 1.4 times faster than serial_bfs. The parallelism of our algorithm, measured in cilkview, is shown in Figure 3.

Second, we attempted to coarsen the algorithm by having each node in the pennant data structure store $v$ vertices
instead of 1. We found that \( v = 64, k = 2 \) yielded the best results, as recommended by Leiserson and Schardl in their paper. Running 5 trials on \( kkt\_power \) on an 8-core cagnode machine yielded an average runtime of 307ms, which was nearly 2 times faster than \( \text{serial\_bfs} \). As shown in Figure 3, this coarsening also yielded high parallelism.

5 Memory Bandwidth Bottleneck

While cilkview showed high parallelism for both coarsened parallelized pennant bag BFS’s, we didn’t achieve significant parallel speedups. We predicted that this was due to insufficient memory bandwidth, since the amount of work performed at each node is relatively small. To test this hypothesis, we increased the arithmetic intensity performed at each memory location. We varied the arithmetic intensity by changing \( \text{extra\_calls} \) where \( \text{extra\_calls} \) is the number of times calling an external library function for each node. The results are shown in Figure 4.

![Figure 4: Varying arithmetic intensity from left to right: \( \text{extra\_calls} = 100, \text{extra\_calls} = 1000, \text{extra\_calls} = 10000. \)](image)

As shown in Figure 4, as \( \text{extra\_calls} \) increases, so do the speedups, which leads us to believe memory bandwidth is in fact the bottleneck.

6 Extensions

For our extension to the lab, we implemented an iterator for more efficiently traversing the bag data structure. Originally, if we needed to traverse the bag data structure, we would convert it into a vector via the \text{copy\_to\_vector} method and then walk through the resulting vector. For our iterator, we instead kept an internal state which allowed us to only copy to a vector the particular pennant that we were currently iterating through. The iterator would keep track of which pennant in the bag the iterator position was currently located at, a copy of the current pennant as a vector, and the position within this vector that the iterator was currently pointing to.

Comparing this implementation to our original one, we observed that the runtimes for the two versions were nearly identical, though the iterator implementation was very slightly faster, averaging 296 ms versus 307 ms. This makes sense since, for the purposes of this lab, every time that an iterator was created it was used to traverse the entire bag, and therefore it does not matter whether the bag is converted into a vector in chunks or all at once. However, for situations in which the iterator would be often required to traverse only a portion of the bag data structure, we would expect such an implementation to be faster and therefore more desirable as a design choice. For instance, in the situation where an iterator is asked to traverse the first \( n/k \) elements of the bag, our implementation would require only \( \Theta(n/k) \) time, as opposed to \( \Theta(n) \) time. Further exploration into the types of situations that would require such traversal patterns could provide interesting test cases for this iterator implementation.

In addition to the iterator, we tried out one of the optimizations discussed in "A Work-Efficient Parallel Breadth-First Search Algorithm." Specifically, we kept track of the highest order pennant within the bag structure. This allowed
us to loop through only the minimum necessary number of pennants, instead of all 64 every time. Since our algorithm entails a large amount of splitting and thus creation of smaller bags, it would be particularly inefficient to loop through all 64 pennant slots for each bag. Thus, we would expect this optimization to save time for any size data set. This optimization was used for all our bag data structures. With more time, we would measure and attempt to quantify the difference that this optimization has on the runtime of our different BFS versions.

Finally, we discovered and documented a few bugs in the pseudo-code of BAG-SPLIT as described in “A Work-Efficient Parallel Breadth-First Search Algorithm.” Specifically:

- Rather than “if $S_1! = \text{NULL}$”, line 4 should read “if $S_1[0]! = \text{NULL}$.”
- “return $S_1$” in line 10 should not be indented.
- Finally, setting $y = S_1[0]$ and then calling $\text{BAG-INSERT}(S_1, y)$ in lines 2 and 10 is incorrect if lines 3-7 do not modify $S_1[k]$. For example, if $S_1[1]$ is null, $S_1[0]$ will not be overridden, and all of the elements of $S_1[0]$ will appear twice in the resulting $S_1$ bag.

Coding, debugging, and writing this report took each of us roughly 18 hours, so extension beyond this point was limited.
Lab4: optimizing bag iterators

For this lab I optimized the bag iterator implementation. For a bag with \( n \) elements and a coarsening factor of \( b \), my most efficient implementation requires \( O(\lg \frac{n}{b}) \) space and moves the iterator to the next element in \( O(1) \) time. My implementation uses software prefetching to optimize performance.

**Overview:** I focused on designing a bag iterator for bags implemented with an array of pennants. When pennants are coarsened with a factor \( b \), each node of the pennant holds an array of elements of size \( b \). At any moment, the position of an individual element in the bag can be represented by three iterators:

- **top** iterator – an array iterator for the current pennant in the array of pennants
- **pennant** (or **middle**) iterator – a binary tree iterator for current node in the current pennant
- **bottom** iterator – an array iterator for the current position in the current node

The bag iterator is implemented using all three iterators.

Figure 1 shows a recursive algorithm for moving an iterator to the next element. The algorithm takes an iterator and recursively tries to move the the next lowest level iterator to the next element (by calling `nextElement`). If this fails, the algorithm tries to update the position of the iterator (by calling `updatePosition`).

The running time of `nextElement` is dominated by the running time of `updatePosition`. The running time of `updatePosition` is the same for the top and bottom iterators. Since the top and bottom iterators iterate over an array, moving to the next element is accomplished by incrementing the position in the array by one. This operation takes \( O(1) \) times and \( O(1) \) space. The design of `updatePosition` for the pennant iterator is more interesting because it requires a non-constant amount of space, which can have a noticeable affect on performance. The next paragraphs describe designs for the middle iterator, starting with an unoptimized naive design.

**Breadth-first iterator:** The breadth-first pennant iterator visits all the nodes in one level of the pennant and records the child nodes to visit next using a FIFO queue. Figure 2 lists the pseudo code for updating the position of a breadth-first pennant iterator. `updatePosition` executes in \( O(1) \) time, but requires \( \Theta(n) \) space, since the queue must be large enough to hold the all the leaves of the pennant, or \( \frac{n}{b} \) elements for a pennant of \( n \) elements. If \( n \) is the number of elements in a bag coarsened with factor \( b \) the space bound is \( O(\frac{n}{b}) \).

**Depth-first iterator:** The depth-first pennant iterator recursively visits all the nodes in the left pennant before visiting nodes in the right pennant. When the left child of a node is visited, the right child added to a LIFO queue, and removed from the queue when iterator has visited the left subpennant. Figure 3 lists the pseudo code for updating the position of the depth-first pennant iterator. `updatePosition` executes in \( O(1) \) time and requires \( \Theta(\lg n) \) space for the LIFO queue when the pennant holds \( n \) elements. The size of the LIFO is determined by the number of nodes the iterator has visited the left children of, but not the right children of. Therefore, the maximum size of the LIFO is the length of the longest path that visits only left children nodes, or \( \lg n \). If \( n \) is the number of elements in a bag coarsened with factor \( b \) the space bound is \( O(\lg \frac{n}{b}) \).

**Optimizations:** For bags with many elements, scarce off-chip DRAM bandwidth is likely to be the limiting performance factor. One way to increase memory bandwidth realized by the application is to use prefetch instructions, which help guide the memory controller on what values to prefetch next. Iterators provide a convenient programming abstraction to implement software prefetching because the details of the software prefetcher policy are hidden from the application.

For a bag implemented using an array of coarsened pennants there exist at least two types of memory accesses that I accelerated using software prefetching. One type is linear accesses that read elements from the coarsened arrays. A linear access is easily predicted by the hardware prefetcher, so software prefetching offers a small benefit for each access. There are many linear accesses while iterating over a bag, however, so the aggregate savings by software prefetching can be quite large. A second type of accesses are the random accesses that occur when walking nodes nodes in the pennant. Random accesses will experience more speedup using software prefetching because they are are not predicted by the hardware prefetcher. For the coarsened bag, however, there are fewer random accesses than linear accesses.

**Performance:** Figure 4 presents the performance of the breadth-first iterator, depth-first iterator, and optimized depth-first iterator. The results are for a microbenchmark that creates a bag containing 100,000,000 8-byte integers.
(approximately 760 Mbytes of data) and sums the contents of the bag using an iterator. The depth-first iterator performs about 15% faster than the breadth-first iterator and the optimized iterator performs about 15% faster than the depth-first iterator. Software prefetching for the linear access and software prefetching for the random accesses contributed equally to the performance improvement of the optimized iterator.

**Miscellaneous:** I was surprised at how much software prefetching helped performance. Potential future work is developing a model for predicting the performance improvements from prefetching. I was also surprised at how transparent software prefetching can be implemented using iterators in C++. More potential future work is using prefetching in other ways and with other abstractions. For example, in some cases an application might not want to cache data it is about to read (*e.g.* a streaming component of an application). An iterator could implement this with the “non-temporal” prefetch instruction supported by recent x86 CPUs.
class Iterator
{
public:
    Iterator(Data *, Pos *);
    bool nextElement(void);

protected:
    bool updatePosition(void);
    Data *data;
    Pos *pos;
    Iterator *lowerIterator;
};

bool Iterator::nextElement(void)
{
    if (lowerIterator) {
        bool ok = this->lowerIterator->nextElement();
        if (!ok && this->updatePosition()) {
            ok = true;
            this->lowerIterator = Iterator(this->data, this->pos);
        }
        return ok;
    }

    // Base case
    return this->updatePosition();
}

Figure 1: The pseudo code for the moving a top, middle, or bottom iterator to the next element.

class BFIterator : public Iterator
{
protected:
    bool updatePosition(void);
    Queue *queue;
    Node *node;
};

bool BFIterator::updatePosition(void)
{
    if (queue.empty()) {
        node = NULL;
        return false;
    }

    queue.pushBack(node->left);
    queue.pushBack(node->right);
    node = queue.popFront();
    return true;
}

Figure 2: The pseudo code for the moving to the next element with a breadth-first pennant iterator.
class DFIterator : public Iterator
{
  protected:
    bool updatePosition(void);
    Queue *queue;
    Node *node;
};

bool DFIterator::updatePosition(void)
{
  if (node->left)
  {
    queue.pushBack(node->right);
    node = node->left;
    return true;
  }
  else
  {
    if (queue.empty()) {
      node = NULL;
      return false;
    }
    node = queue.popBack();
  }
  return true;
}

Figure 3: The pseudo code for moving to the next element with a depth-first pennant iterator.

<table>
<thead>
<tr>
<th>Time (secs)</th>
<th>BF iterator</th>
<th>DF iterator</th>
<th>Opt. DF iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.8</td>
<td>2.4</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Figure 4: The performance of the breadth-first iterator, depth-first iterator, and optimized depth-first iterator.
6.884 Lab 4 – Breadth-first Search

Nathan Beckmann

March 18, 2010

1 Serial Breadth-first Search

The queue in Figure 1 never contains nodes from more than two layers because of the coloring. We can show this inductively. Initially, the queue contains only the start node, so the condition is satisfied.

Now we must show the invariant that given the head of the queue is in layer $k$, the queue only contains nodes from layers $k$ and $k+1$. This is true because an entire layer is added to the queue before the next, that is all nodes in layer $k$ appear before all nodes in layer $k+1$. This is true because in step 12 we process only adjacent nodes that are colored white (meaning they have not been previously processed). So it is impossible for a node in layer $1...k−1$ to appear in the queue. By similar reasoning, it is impossible for a node in layer greater than $k+1$ to appear, because we only process nodes adjacent to a node in layer $k$. By definition, this can only be in a layer $\in \{k−1,k,k+1\}$. But we already said that it hasn’t been processed, so by induction hypothesis it must be in layer $k+1$.

My code for a 2 queue version of BFS is in the appendix (separate document submitted on Stellar). I essentially keep two queues: one for the current layer being processed, and one for the next. The variable curq tracks which one is active, and I reset the head/tail to the beginning of the queue at each layer. Its runtime is very similar (slightly slower?) than the previous version.

2 BFS with Bags

Note: All numbers in this section are gathered with the kkt_power.bin input.

The code for the bag reducer is listed in the appendix. This code performs slightly worse than the simple queue implementation, as all we’ve done here really is add some overhead with linked lists to a simple vector implementation.

In order to implement this in parallel, I separated my function to process a layer into a base case function and a recursive function.

The speed up I get from this is pretty bad (see Figure 1). This is for at least three reasons: (i) the split() function provided by the BagList object does a bad job of evenly dividing work, giving poor parallelism, (ii) the BagList object splits at a small grain size (I manually increased this from 10 to 1000 to improve
things slightly), and (iii) the application is bandwidth intensive, as discussed in class. This last point is demonstrated by running independent, serial copies of the application:

<table>
<thead>
<tr>
<th>Number of copies</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.3</td>
</tr>
<tr>
<td>2</td>
<td>2.9</td>
</tr>
<tr>
<td>3</td>
<td>4.1</td>
</tr>
<tr>
<td>4</td>
<td>5.9</td>
</tr>
</tbody>
</table>

Table 1: Run-times of independent, serial runs.

As you can see, runtime of each copy increases almost perfectly linearly with the number of copies being run, meaning system throughput is constant. This indicates that system throughput is somehow limited by I/O – in this case, memory bandwidth.

The race on this code is in adding a node to the queue to be processed next time (next_layer in my code). This means that sometimes we will process a node more than once, generating extra work. This is not a problem with correctness, as the code will still produce the correct distances on each node. But it is suboptimal, as if this happens early in the program than one can imagine exponentially increasing work as nodes are added branching out from the first node. But this is extremely unlikely, as when the node is processed a second time, the races must occur again in order for its children to be added. So in this sense, the code is self correcting and we shouldn’t expect too much extra work to be generated from the race.
3 Parallel Bag (Pennants)

I implemented a non-coarsened Pennant structure and used that to implement a Bag. I found that it was much slower in the serial case than previous algorithms, taking 3.3 sec to run, as opposed to 0.6 or 1.3 sec for straightforward BFS or the BagList. Figure 2 shows that it gets much better speedup, however. While the speedup is not ideal, at least is still increases as cores are added and seems linear. Code is given in the appendix.

![Speedup Chart](image)

Figure 2: Speed-up of parallel BFS using pennant-based Bag implementation.

I might expect this code to scale poorly because it should be fairly memory heavy, especially with all of the pointers being chased in the tree structures. But I do see decent scaling with this version, probably because it is fairly slow which tends to improve scalability.

4 Optimized Parallel Bag

I optimized my implementation of a parallel bag by coarsening the leaves of the tree to vectors of size 4k. This ended up being the fastest implementation of BFS, taking only 0.214 sec to complete on 8 cores. Furthermore, on a single core, it takes 0.592 sec, which is again faster than the original serial implementation. So it seems that this Bag implementation is able to overcome the overhead of using a more complicated data structure and provide real speedup.

One nice thing about this implementation is that my Pennant implementation was completely unmodified. I just had to change a few methods in the Bag class, and even these were largely minor changes. The biggest change was in the `insert` method and dealing with the `Iterator` class.

Part of the reason this might work well is that I did one tweak to my bag
implementation to improve performance. To optimize performance of iterators, the bag caches the most recently accessed leaf node. This gives very fast lookup into the vector and eliminates the need to traverse the tree in order to find the data. I believe this was a big part of my overhead previously. For example, this is the implementation of dereference: (full listing given in appendix)

```cpp
const T& dereference(BaseIterator* it) const
{
  if (_cachedPennantNum == it->_pennantNum &&
      _cachedPennantOffset == it->_pennantOffset)
  {
    return _cachedVec->at(it->_vecOffset);
  }
  else
  {
    _cachedPennantNum = it->_pennantNum;
    _cachedPennantOffset = it->_pennantOffset;
    _cachedVec = &_pennants[it->_pennantNum]->at(it->_pennantOffset);
    return _cachedVec->at(it->_vecOffset);
  }
}
```

Figure 3 shows that while we see some speedup going across cores, it isn’t as much as in the unoptimized case. This is probably because:

- The unoptimized version can split at finer granularity, and therefore exposes more parallelism.
- More importantly, the unoptimized version is slow, and it is always easier to demonstrate scalability with bad code.

5 Adding Work

I added extra work to each computation by adding the following to the inner loop of the bfs_walk_layer_base function, which is the base of my recursion:

```cpp```
for (int i = 0; i < 0xFFF; i++)
  trivial();
```

This slowed things down significantly – it went from running in .5 sec to 116 sec. But the scaling improved tremendously, as seen in Figure 4.

6 Further Exploration

For this lab, I decided to analyze the performance of my Iterator class. As mentioned above, I optimized this iterator to be cached by the Bag to improve
performance. To sanity check this, I removed the caching and re-ran the numbers. I found that there was no difference in performance with caching. So my original assumption was completely unfounded. With this in mind, what is the average access time to an iterator? It clearly can’t be much, as caching doesn’t help at all.

Let \( n \) be the number of items in a bag. Let \( v \) be the size of the vector at the leaves. Then there are \( n/v \) nodes in the bag, and \( \log_2 n/v \) pennants in the bag. Referring to the appendix (code listing), advancing an iterator in the worst case requires traversing the list of pennants, traversing the pennant to find the leaf node, and performing a lookup in the vector. Traversing the pennants takes \( \Theta(\log_2 n/v) \) time. Traversing the pennant on the order of the height of the pennant, or \( \log_2 n/2v = \log_2 n/v - 1 \) (the size of the largest pennant is bounded above by \( n/2v \)). Looking up in the vector takes constant time, so we have:

\[
O(\text{advance}) = 2 \log_2 \frac{n}{v} = O \left( \log_2 \frac{n}{v} \right)
\]

The time to traverse the entire bag is given by the total time to traverse the bag divided by \( n \). First, it will take \( \Theta(\log_2 n/v) \) time to traverse the list of pennants.

Now consider a pennant of height \( h \). This pennant contains \( 2^h \) nodes and \( v2^h \) items. Each item takes \( \Theta(h) \) to find the leaf, and \( \Theta(1) \) time to lookup. So the total time to traverse the pennant is

\[
\Theta(v2^h \cdot h)
\]

If \( n_i \) is the \( n^{th} \) bit of \( n \) as a binary integer, the amortized running time of
Figure 4: Speed-up of parallel BFS when work is artificially added.

The advance is

\[ \Theta \left( \sum_{i=0}^{\log_2 \frac{n}{v}} n_i \frac{n}{v}^i + \log_2 \frac{n}{v} \right) \]

Note that this value depends on the \( n_i \) — that is the internal structure of the pennants in the bag. First consider the worst-case for the amortized running time, when all \( n_i = 1 \). In this case,

\[
O \left( \frac{v n \log_2 \frac{n}{v} + \log_2 \frac{n}{v}}{n} \right)
= O \left( \log_2 \frac{n}{v} \right)
\]

To sanity check this, if \( n = v \), then the amortized advance time is \( \Theta(1) \), which seems correct. Similarly, if \( v = 1 \), then the amortized time is \( \log_2 n \), which is simply the time of going through a pennant of size \( \Theta(n) \) (the time to traverse the pennants or the smaller pennants disappears because the number of items in each pennant increases exponentially).

From this result, we should expect that the best-case amortized running time should be \( \Theta(\log_2 n/v) \) as well. The intuition for this is that \( n_{\log_2 n/v} = 1 \) always (by definition), so the largest possible pennant will always be present. If the largest pennants dominate amortized run-time in the worst case, and the best-case always has the largest pennants full, we should expect the same amortized run-time. The analysis justifies this:

\[ \Omega(\text{advance}) = \left( \frac{v \frac{n}{v} \log_2 \frac{n}{v} + \log_2 \frac{n}{v}}{n} \right) \]
\[ = \Omega \left( \log_2 \frac{n}{v} \right) \]

Since the amortized run-time is bounded above and below by \( \log_2 \frac{n}{v} \), this is the amortized run-time. These results indicate that the amortized and worst-case performance for the pennant-based bag are the same (up to a constant factor). This is a cool result.
6.884 Lab 4 – Breadth-first Search
Code Listings

Nathan Beckmann
March 18, 2010

This appendix contains code listings for all of the interesting code in this lab.

1  bfs_2queue Implementation

```
int
Graph::bfs_2queue(const int s, unsigned int *distances)
{
    unsigned int *q[2];
    unsigned int curq;
    unsigned int head, size, tail;
    unsigned int current, newdist;

    if (s < 0 || s > (int)nNodes)
        return -1;

    q[0] = new unsigned int[nNodes];
    q[1] = new unsigned int[nNodes];

    for (unsigned int i = 0; i < nNodes; i++) {
        distances[i] = D_INFINITY;
    }

    curq = 0;
    current = s;
    distances[s] = 0;
    head = 0;
    size = 1;
    tail = 0;
    while (1) {
        for (head = 0; head < size; head++)
```
newdist = distances[current]+1;
for (int i = nodes[current]; i < nodes[current+1]; i++) {
    if (newdist < distances[edges[i]])
        q[!curq][tail++] = edges[i];
    distances[edges[i]] = newdist < distances[edges[i]] ?
        newdist : distances[edges[i]];
}
current = q[curq][head];
}

if (tail == 0)
    break;

curq = !curq;
head = 0;
size = tail;
tail = 0;
}
delete q[0];
delete q[1];

return 0;

2 Serial BFS with Bags

template <typename T, BagType BType>
static int bfs_walk_layer(Bag_reducer<T, BType> & this_layer_reducer, 
                        Bag_reducer<T, BType> & next_layer_reducer, unsigned int * distances, 
                        int * nodes, unsigned int nNodes, int * edges, unsigned int nEdges)
{
    unsigned int current, newdist;
    typename Bag<T, BType>::BaseIterator it;
    Bag<T, BType> & this_layer = this_layer_reducer.get_reference();

    for (it = this_layer.begin(); it != this_layer.end(); this_layer.advance(&it))
    {
        current = this_layer.dereference(&it);

        newdist = distances[current]+1;
        for (int i = nodes[current]; i < nodes[current+1]; i++) {
            if (newdist < distances[edges[i]])
                next_layer_reducer.insert(edges[i]);
        }
    }

    return 0;
}
distances[edges[i]] = newdist < distances[edges[i]] ?
    newdist : distances[edges[i]];
}
}

return 0;
}

int Graph::bfs_bag_list(const int s, unsigned int *distances)
{
    Bag_reducer<unsigned int, BAG_LIST> q[2];
    unsigned int curq;

    if (s < 0 || s > (int)nNodes)
        return -1;

    for (unsigned int i = 0; i < nNodes; i++) {
        distances[i] = D_INFINITY;
    }

    curq = 0;
    q[curq].insert(s);
    distances[s] = 0;

    while (!q[curq].is_empty()) {
        bfs_walk_layer(q[curq], q[!curq], distances, nodes, nNodes,
                        edges, nEdges);
        q[curq].clear();
        curq = !curq;
    }

    return 0;
}

3  Parallel BFS with Bags

template <typename T, BagType BType>
static int bfs_walk_layer_base(Bag<T, BType> & this_layer, Bag<T, BType> & next_layer, unsigned int *distances, int *nodes, unsigned int nNodes, unsigned int nEdges)
{
    unsigned int current, newdist;
    typename Bag<T, BType>::BaseIterator it;

    for (it = this_layer.begin(); it != this_layer.end(); ++it) {
        current = *it;
        newdist = 0;

        for (it = next_layer.begin(); it != next_layer.end(); ++it) {
            if (*it == current) {
                newdist = distances[current] + 1;
                distances[*it] = newdist < distances[*it] ?
                    newdist : distances[*it];
            }
        }

        if (newdist < distances[current]) {
            distances[current] = newdist;
            q[curq].insert(current);
        }
    }

    return 0;
}
for (it = this_layer.begin(); it != this_layer.end();
    this_layer.advance(&it))
{
    current = this_layer.dereference(&it);
    
    newdist = distances[current] + 1;
    for (int i = nodes[current]; i < nodes[current + 1]; i++) {
        if (newdist < distances[edges[i]])
            next_layer.insert(edges[i]);
        
        distances[edges[i]] = newdist < distances[edges[i]] ?
            newdist : distances[edges[i]];
    }
}

return 0;
}

template <typename T, BagType BType>
static int bfs_walk_layer(Bag<T, BType> & this_layer, Bag<T, BType> &
next_layer, unsigned int * distances, int * nodes, unsigned int
nNodes, int * edges, unsigned int nEdges)
{
    if (this_layer.can_split())
    {
        Bag<T, BType> * this_split = this_layer.split();
        Bag<T, BType> next_split;
        cilk_spawn bfs_walk_layer(*this_split, next_split, distances,
        nodes, nNodes, edges, nEdges);
        bfs_walk_layer(this_layer, next_layer, distances, nodes,
        nNodes, edges, nEdges);
        cilk_sync;
        next_layer.merge(&next_split);
        delete this_split;
    }
    else
    {
        bfs_walk_layer_base(this_layer, next_layer, distances, nodes,
        nNodes, edges, nEdges);
    }

    return 0;
}
4 BagParallel Implementation with Pennants

template <typename T>
class Pennant
{
public:

    Pennant()
    : _size(-1)
      , _root(NULL)
    {}
    Pennant(const T &d)
    : _size(-1)
      , _root(NULL)
    {
        insert(d);
    }
    ~Pennant()
    {
        clear();
    }

class TreeNode
{
public:
    TreeNode(bool leaf)
    : _leaf(leaf)
    {}

    virtual ~TreeNode()
    {
        bool _leaf;
    }

class TreeDataNode : public TreeNode
{
public:
    TreeDataNode(const T & d)
    : TreeNode(true)
      , _data(d)
    {
    } T _data;

class TreeBranchNode : public TreeNode
{
public:
    TreeBranchNode(TreeNode * left, TreeNode * right)
    : TreeNode(false)
~TreeBranchNode()
{
  delete _left;
  delete _right;
}
TreeNode * _left;
TreeNode * _right;
};
Pennant<T> * merge(Pennant * rhs)
{
  Pennant<T> * out = new Pennant();
  out->_root = new TreeBranchNode(this->_root, rhs->_root);
  out->_size = this->_size + 1;
  this->_root = NULL; this->_size = -1;
  rhs->_root = NULL; rhs->_size = -1;
  return out;
}
void split(Pennant<T> ** left,
            Pennant<T> ** right)
{
  TreeBranchNode * rootb = (TreeBranchNode*)_root;

  *left = new Pennant<T>();
  *right = new Pennant<T>();

  (*left)->_root = rootb->_left;
  (*right)->_root = rootb->_right;

  (*left)->_size = (*right)->_size = this->_size - 1;

  rootb->_left = NULL; rootb->_right = NULL;
  clear();
}
void insert(const T & data)
{
  clear();
  _root = new TreeDataNode(data);
  _size = 0;
}
void clear()
{
  _size = -1;
  delete _root;
}
bool empty()
{
    return _root == NULL;
}
const T & at(int offset)
{
    TreeNode * n = _root;
    for (int i = 0; i < _size; i++)
    {
        TreeBranchNode * b = (TreeBranchNode*) n;
        if (offset & (1 << i))
            n = b->_left;
        else
            n = b->_right;
    }
    return ((TreeDataNode*)n)->_data;
}

int _size;
TreeNode * _root;

template <typename T>
class Bag<T, BAG_PARALLEL> { 
    public:
    Bag<T, BAG_PARALLEL>()
    { }
    Bag<T, BAG_PARALLEL>(Bag<T, BAG_PARALLEL>*)
    { assert(false); }
    ~Bag<T, BAG_PARALLEL>()
    { for (size_t i = 0; i < _pennants.size(); i++)
        { delete _pennants[i]; }
    }
    void insert(T d)
    { insert(new Pennant<T>(d)); }
    void insert(Pennant<T> * p)
    { if (p == NULL)
return;

if ((size_t)p->_size+1 > _pennants.size())
{
    size_t oldsize = _pennants.size();
    _pennants.resize(p->_size+1);
    for (size_t i = oldsize; i < _pennants.size(); i++)
        _pennants[i] = NULL;
}

if (_pennants[p->_size] == NULL)
{
    _pennants[p->_size] = p;
}
else
{
    Pennant<T> * p2 = _pennants[p->_size];
    _pennants[p->_size] = NULL;
    insert(p->merge(p2));
    delete p;
    delete p2;
}
}

void merge(Bag<T, BAG_PARALLEL> * b)
{
    for (size_t i = 0; i < b->_pennants.size(); i++)
    {
        insert(b->_pennants[i]);
        b->_pennants[i] = NULL;
    }
}

Bag<T, BAG_PARALLEL>* split(void)
{
    Bag<T, BAG_PARALLEL> * b = new Bag<T, BAG_PARALLEL>();
    std::vector<Pennant<T> * > newPennants;
    b->_pennants.resize(_pennants.size() - 1);
    newPennants.resize(_pennants.size());
    for (size_t i = 1; i < _pennants.size(); i++)
    {
        if (_pennants[i] == NULL)
        {
            b->_pennants[i-1] = NULL;
            newPennants[i-1] = NULL;
        }
        else
        {

8
b->_pennants[i-1] = NULL;
_pennants[i]->split(&(b->_pennants[i-1]), &(newPennants[i-1]));
}
delete _pennants[i];
}
Pennant<T> * one = _pennants[0];
_pennants.assign(newPennants.begin(), newPennants.end());
insert(one);
return b;
}

bool is_empty() const
{
  for (size_t i = 0; i < _pennants.size(); i++)
  {
    if (_pennants[i] != NULL)
      return false;
  }
  return true;
}

int num_elements() const
{
  int n = 0;
  for (size_t i = 0; i < _pennants.size(); i++)
  {
    if (_pennants[i] != NULL)
    {
      n += (1 << i);
    }
  }
  return n;
}

void clear()
{
  for (size_t i = 0; i < _pennants.size(); i++)
  {
    delete _pennants[i];
    _pennants[i] = NULL;
  }
}

bool can_split(void) const
{
  for (size_t i = 1; i < _pennants.size(); i++)
  {
    if (_pennants[i] != NULL)
    {
      return true;
    }
  }
  return false;
}
return true;
}
return false;
}
typename std::vector<T>* copy_to_vector() const
{
    std::vector<T> * vec = new std::vector<T>();
    vec->reserve(num_elements());
    Iterator i = begin();
    while (i != end())
    {
        vec->push_back(dereference(&i));
        advance(&i);
    }
    return vec;
}
struct Iterator
{
    int _pennantNum;
    int _offset;
    Iterator(int p, int o)
        : _pennantNum(p)
          , _offset(o)
    {
    }
    bool operator==(const Iterator & it) const
    {
        return _pennantNum == it._pennantNum &&
               _offset == it._offset;
    }
    bool operator!=(const Iterator & it) const
    {
        return !(*this == it);
    }
};
typedef Iterator BaseIterator;
BaseIterator begin() const
{
    Iterator it(-1,0);
    advance(&it);
    return it;
}
BaseIterator end() const
{
    return Iterator(_pennants.size(),0);
    size_t i;
for (i = _pennants.size() - 1; i > 0; i--)
{
    if (_pennants[i])
        break;
}
return Iterator(i, 1 << i);
}

void advance(BaseIterator* it) const
{
    it->_offset++;
    if (it->_offset >= (1 << it->_pennantNum))
    {
        while (it->_pennantNum < (int)_pennants.size() && _pennants[++it->_pennantNum] ==
            ;
        it->_offset = 0;
    }
}

const T& dereference(BaseIterator* it) const
{
    return _pennants[it->_pennantNum]->at(it->_offset);
}

private:
    std::vector<Pennant<T> *> _pennants;
};

5 Optimized Bag Implementation

template <typename T>
class Bag<T, BAG_OPT> {

public:

    static const int GRAIN_SIZE = 4096;

    struct TVec
    {
        TVec()
            : _size(0)
        {
        }
        ~TVec()
        {
        }

        size_t size() const { return _size; }
        size_t capacity() const { return GRAIN_SIZE; }
        void push_back(T t) { _data[_size++] = t; }
    }
const T & at(size_t index) const { return _data[index]; }

private:
    size_t _size;
    T _data[GRAIN_SIZE];
};

// The common methods for the bag interface.
Bag<T, BAG_OPT>()
    : _cachedPennantNum(-1)
    , _cachedPennantOffset(-1)
    , _cachedVec(NULL)
{ }
Bag<T, BAG_OPT>(Bag<T, BAG_OPT>*)
{ assert(false); }
"Bag<T, BAG_OPT>()
{ for (size_t i = 0; i < _pennants.size(); i++)
    { delete _pennants[i]; }
}
void insert(T d)
{ TVec * tv;
    if (!._pennants.empty() && _pennants[0])
        tv = &((typename Pennant<TVec>::TreeDataNode*)._pennants[0]->_root)->_data;
    else
        tv = NULL;

    if (tv && tv->size() < tv->capacity())
        tv->push_back(d);
    else
    { // It is ugly to manually fill in the nodes here, but
        // it is faster than copying the entire vector...
        Pennant<TVec> * p = new Pennant<TVec>();
        typename Pennant<TVec>::TreeDataNode * n = new typename Pennant<TVec>::TreeDataNode();
        n->_data.push_back(d);
        p->_root = n;
        p->_size = 0;
        insert(p); }
}  

void insert(Pennant<TVec> * p) 
{
    if (p == NULL)
        return;

    if ((size_t)p->_size + 1 > _pennants.size())
    {
        size_t oldsize = _pennants.size();
        _pennants.resize(p->_size + 1);
        for (size_t i = oldsize; i < _pennants.size(); i++)
            _pennants[i] = NULL;
    }

    if (_pennants[p->_size] == NULL)
    {
        _pennants[p->_size] = p;
    }
    else
    {
        Pennant<TVec> * p2 = _pennants[p->_size];
        _pennants[p->_size] = NULL;
        insert(p->merge(p2));
        delete p;
        delete p2;
    }
}

void merge(Bag<T, BAG_OPT> * b) 
{
    for (size_t i = 0; i < b->_pennants.size(); i++)
    {
        insert(b->_pennants[i]);
        b->_pennants[i] = NULL;
    }
}

Bag<T, BAG_OPT>* split(void)
{
    Bag<T, BAG_OPT> * b = new Bag<T, BAG_OPT>();
    std::vector<Pennant<TVec> * > newPennants;
    b->_pennants.resize(_pennants.size() - 1);
newPennants.resize(_pennants.size());

for (size_t i = 1; i < _pennants.size(); i++)
{
    if (_pennants[i] == NULL)
    {
        b->_pennants[i-1] = NULL;
        newPennants[i-1] = NULL;
    }
    else
    {
        b->_pennants[i-1] = NULL;
        _pennants[i]->split(&(b->_pennants[i-1]), &(newPennants[i-1]));
    }

delte _pennants[i];
}

Pennant<TVec> * one = _pennants[0];

_pennants.assign(newPennants.begin(), newPennants.end());

insert(one);

return b;
}

bool is_empty() const
{
    for (size_t i = 0; i < _pennants.size(); i++)
    {
        if (_pennants[i] != NULL)
            return false;
    }
    return true;
}

int num_elements() const
{
    int n = 0;

    for (size_t p = 0; p < _pennants.size(); p++)
    {
        if (_pennants[p] == NULL)
            continue;
    }
for (size_t i = 0; i < (1u << p); i++)
    n += _pennants[p]->at(i).size();

return n;
}

void clear()
{
    for (size_t i = 0; i < _pennants.size(); i++)
    {
        delete _pennants[i];
        _pennants[i] = NULL;
    }
}

bool can_split(void) const
{
    for (size_t i = 1; i < _pennants.size(); i++)
    {
        if (_pennants[i] != NULL)
            return true;
    }

    return false;
}

typename std::vector<T>* copy_to_vector() const
{
    std::vector<T> * vec = new std::vector<T>();

    Iterator i = begin();
    while (i != end())
    {
        vec->push_back(dereference(&i));
        advance(&i);
    }

    return vec;
}

struct Iterator
{
    int _pennantNum;
    int _pennantOffset;
    int _vecOffset;
Iterator(int p, int o, int v)
    : _pennantNum(p)
    , _pennantOffset(o)
    , _vecOffset(v)
{}

bool operator==(const Iterator & it) const
{
    return _pennantNum == it._pennantNum &&
    _pennantOffset == it._pennantOffset &&
    _vecOffset == it._vecOffset;
}

bool operator!=(const Iterator & it) const
{
    return !(this == it);
}

};

typedef Iterator BaseIterator;
BaseIterator begin() const
{
    Iterator it(-1,0,GRAIN_SIZE);
    advance(&it);
    return it;
}
BaseIterator end() const
{
    return Iterator(_pennants.size(),0,0);
}
void advance(BaseIterator* it) const
{
    // there has to be a shorter way to do this ...
    const TVec * tv;

    if (_cachedPennantNum != it->_pennantNum ||
        _cachedPennantOffset != it->_pennantOffset)
    {
        _cachedPennantNum = it->_pennantNum;
        _cachedPennantOffset = it->_pennantOffset;
        tv = _cachedVec;
    }
    else
    {
        if (it->_pennantNum >= 0 && _pennants[it->_pennantNum])
            tv = _pennants[it->_pennantNum]->at(it->_pennantOffset);
        else
tv = NULL;
}

++it->_vecOffset;
if (tv == NULL || (size_t)it->_vecOffset >= tv->size())
{
  it->_vecOffset = 0;

  it->_pennantOffset++;
  if (it->_pennantOffset >= (1 << it->_pennantNum))
  {
    while (it->_pennantNum < (int)_pennants.size() && _pennants[++it->_pennantNum])
      ;

    it->_pennantOffset = 0;
  }
}

_cachedPennantNum = it->_pennantNum;
_cachedPennantOffset = it->_pennantOffset;
if ((size_t)it->_pennantNum < _pennants.size() && _pennants[it->_pennantNum])
  _cachedVec = &_pennants[it->_pennantNum]->at(it->_pennantOffset);
}

const T& dereference(BaseIterator* it) const
{
  if (_cachedPennantNum == it->_pennantNum &&
      _cachedPennantOffset == it->_pennantOffset)
  {
    return _cachedVec->at(it->_vecOffset);
  }
  else
  {
    _cachedPennantNum = it->_pennantNum;
    _cachedPennantOffset = it->_pennantOffset;
    _cachedVec = &_pennants[it->_pennantNum]->at(it->_pennantOffset);
    return _cachedVec->at(it->_vecOffset);
  }
}

private:
  mutable int _cachedPennantNum;
  mutable int _cachedPennantOffset;
  mutable const TVec * _cachedVec;

  std::vector<Pennant<TVec> *> _pennants;
};
Stencil Computing

Summary

In this lab, I look at parallelizing the breath-first-search (BFS) algorithm, an algorithm that explores a graph outward from a single vertex visiting nodes in increasing order of their distance from the source node. The key insights to the BFS computation are two-fold: First, we can process each layer in parallel since the exact order that we discover the nodes won’t affect the computed distance. Second, redoing some work will not compromise the correctness of the algorithm. The following sections describe my approach:

- Section 1 describes the high-level implementation.
- Section 2 describes the implementation of a more efficient hyper-object to speed up the parallelization.
- Section 3 looks at the amount of repeated work that the implementation does and looks at addressing this with the atomic operations provided by C++.

1 High Level

At the high-level the implementation follows the standard BFS algorithm as described in CLRS. The algorithm maintains a queue of the vertices that it is processing. At each step, it pulls 1 vertex off the queue, and add all of its neighbors that it has not yet seen to the end of the queue after updating their distance from the source based on the distance to the current node and the distance from that node to its neighbor (in this lab I consider only graphs which have edge weights of 0 or 1).

Because BFS visits nodes in order of increasing distance from the source, I can break the queue into two queues, one that the algorithm is processing from which contains the vertices distance $k$ from the source and another which has the vertices of distance $k + 1$. By breaking the queue, the algorithm can consider the from queue in batch, thereby gaining some parallelism. The following are the algorithmic variants based in this insight, Figure 1 gives the speedups of these variants, and Figure 2 gives the scalability analysis of the bag-vector implementation

serial The traditional, 1 queue, serial, implementation.

2queue A serial implementation that uses 2 queues.

bag-vector Parallel implementation of the two queue algorithm. The from queue is processed in parallel by converting it to a vector and then using a cilk_for loop.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1.39</td>
<td>–</td>
</tr>
<tr>
<td>2queue</td>
<td>1.29</td>
<td>1.08</td>
</tr>
<tr>
<td>bag-vector</td>
<td>1.13</td>
<td>1.23</td>
</tr>
</tbody>
</table>

Figure 1: Speedups for the basic algorithm variants.

Both variants contribute some speedup to the basic serial version. I can’t really explain the 2queue variant behavior, but the bag-vector speedup is due to the parallelism, though running on 8 workers and 8 cores, this is a terrible speedup. The scalability results look in more depth at this result.

The scalability of the bag-vector is pretty bad. In addition to hugging the lower-bound for burdened parallelism, the horizontal line at 3 suggests that it isn’t possible in the model to get more than about 3.8x speedup on this code,
2 A More Efficient Reducer

I improve the reducer by implementing sets using bags of pennants. A pennant is simply a binary tree that has an extra node at the top which makes merging two pennants easier. The bag keeps an array of these pennants such that the index in the array corresponds to the size of the pennant, i.e. the \( i^{th} \) cell has a pennant of size \( 2^i \). I compare two new variants to the previous bag-vector variant described in Section 1.

**bag** The multi-set reducer implemented as a bag of pennants. One data item is stored at each pennant point.

**coarse-bag** The same as the **bag** variant except that each pennant stores multiple values and the bag itself has an additional array to store values before there are enough to fill a pennant slot.

Surprisingly, the **bag** variant is actually slower than the **bag-vector** variant. This could be due to the book-keeping that the **bag** implementation is doing. It could also be contributed to by the traversal algorithm not being coarsened sufficiently, though a few attempts to tune the coarsening were unsuccessful in speeding up the computation. The results for the **coarse-bag** variant suggest that the overhead of the **bag** variant is a considerable part of the computation.

The scalability analysis shows that the bag results have much better theoretical scalability than the bag-vector variant, though their speedup curves are still very low. I’m not entirely sure why the burdened parallelism of the coarse-bag variant falls off more at the top than the simple **bag** variant. My only thought here is that it is due to memory allocation overhead since this algorithm allocates considerably more memory than the **bag** algorithm.

3 Extra Work & Atomic Operations

I wanted to look at the amount of extra work that was being done, and how it relates to the number of processors. Figure 5 shows the average number of additional nodes visited over 16 runs with the different worker counts for each graph.
Figure 4: Scalability analysis for the bag reducer variants run on 8 cores using the vanHeukelum-cage15 graph.

<table>
<thead>
<tr>
<th>graph</th>
<th>Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>layered</td>
<td>4835.75</td>
</tr>
<tr>
<td>kkt_power</td>
<td>3.13</td>
</tr>
<tr>
<td>vanHeukelum-14</td>
<td>10.25</td>
</tr>
<tr>
<td>vanHeukelum-15</td>
<td>8.88</td>
</tr>
</tbody>
</table>

Figure 5: Extra work done by the coarse-bag algorithm.
I measured extra work by modifying the traversal algorithm to perform atomic compare and swap using `__sync_bool_compare_and_swap`\(^1\). Because the amount of extra work done is so small, and probably also because the algorithm is memory-bound, the overhead paid by the atomic operation is significant. At the processor level, the cores are doing at least a memory fence operation here, but probably also need to do some inter-core locking.

It seems like, at least for this problem, the atomic operations don’t pay off, but they might be of some as an alternative to heavier locks for some problems. The performance degradation is given in Figure 6. The numbers actually suggest that the atomic operation is very bad for the processor memory unit. On the layered graph, the one with almost 50% more work done in this case, actually slows down the most. It would be interesting to look at the corresponding numbers for mutexes, but I didn’t have time to investigate this farther.

---

\(^1\)I compared with running the same code using a single thread and the numbers were the same.
1 Overview

In this lab I parallelized breadth first search using 3 versions of the bag data structure: linked-list bag (provided to us), Pennant bag, and a coarsened Pennant bag. The linked-list bag performed very poorly due to its inefficient split method; however, the two parallel versions of the pennant bag data structures ran faster, beating the serial queue and the 2-queue implementations. The running times are presented in Table 1 below (the parallel code ran on 8 workers).

Table 1: Serial and Parallel Running Times (s). Notation: Q = single queue; 2-Q = two queues; L-L= linked-list; Crs = coarsened.

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>2-Q</th>
<th>L-L (Ser)</th>
<th>L-L (Par)</th>
<th>Pennant (Ser)</th>
<th>Pennant (Par)</th>
<th>Crs-Pennant (Ser)</th>
<th>Crs-Pennant (Par)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.61</td>
<td>0.615</td>
<td>33.572</td>
<td>82.324</td>
<td>1.418</td>
<td>0.517</td>
<td>0.885</td>
<td>0.358</td>
</tr>
</tbody>
</table>

2 Pennant Bag

I implemented the Pennant data structure (a complete binary tree with an extra root node) as a tree of PennantNodes. A PennantNode stores a value and pointers to its left and right child. The Pennant stores the root of the tree and the total number of elements in the tree. The merge and split methods were implemented as described in lecture/handout.

The Pennant bag keeps a vector of pointers to Pennants. The vector is initialized to a size of 20 (which is enough for the size of our graph). The insert and merge procedures were implemented as described in the lab write-up. In order to split a bag, we can just iterate over all the pennants in the vector (except the first one), splitting each pennant at vector index k and setting the vector value at k − 1 in each bag to one of the split results. The first pennant of size 1 can then be inserted into one of the bags. This split function is much more balanced: each pennant splits exactly in half and the only difference between bag sizes is at most one element.

Pennant Bag Iterator. We traverse the bag data structure using the PennantBagIterator. This iterator is composed of two internal iterators: a PennantIterator over the pennant trees and a vector iterator over the bag vector that stores the pennants. The vector iterator traverses the bag looking for a non-zero pennant pointer. When such a pointer is found, it creates a PennantIterator giving it the root to the found pennant. Upon advance, if we are currently traversing a pennant, we just advance the PennantIterator to the next node, when we are done with a pennant, we advance the vector iterator to the next available pennant. The end() of the bag iterator is equivalent to the end() of the vector storing the pennants.

Pennant Iterator. In order to traverse the tree more efficiently, we can keep some state at the iterator. We keep a pointer to the current node (whose value should be returned by a dereference) and a list of nodes we
should traverse next. Elements are added to this list as we traverse the pennant in the following way. When
the iterator advances, it pops a node from the front of the list and sets it as the current node, it then adds the
left and right child of this node (if they exist) to the list. When the list is empty, we know we have traversed
the entire pennant.

3 Coarsened Pennant Bag

It is not efficient to only store one element per node; therefore, I coarsened the leaves of the pennants, allowing
more than one value to be stored in each leaf. The LEAF_CAPACITY constant determines how many nodes
can be kept in a leaf. In order to achieve that, I modified the PennantNode to store a vector of elements when
it is a leaf (otherwise it only stores one element). The leaves of all the pennants composed of more than one
node are filled to capacity – the pennant with 1 node is a leaf node which can have up to LEAF_CAPACITY
elements.

Insert, Merge, Split (Updated). To insert into a bag, we do something similar to the old insert procedure
except for the additional initial check: if the leaf capacity of the one-node pennant (i.e. pennant at position
0 in the vector) is not reached, we can just insert the element into this leaf, otherwise, we create a two-node
pennant and insert recursively as usual. In order to merge two bags, we need to take care of the two single
leaf nodes - we can just insert all the elements from the leaf node of the argument bag into our final bag.
The split call can also stay mostly the same – we split every pennant with more than 1 node and just insert
the remaining leaf’s elements into one of the bags. When we split the pennant of 2 nodes, we end up with
two leaf nodes – one with full capacity and one with only one element (we insert the elements into the
one-element leaf). This split function is not as balanced as the one of the one-element-per-node pennant;
however, the differences between bag sizes are not too large for reasonable LEAF_CAPACITY values.

PennantIterator (Updated). The pennant iterator needs to differentiate between the case when its current
node is a leaf or not. If the node is a leaf, we need to iterate over all its elements before proceeding to the
next node. We can keep an index into the leaf vector, which gets assigned to a valid value when we iterate
over a leaf.

Alternative. A probably better alternative would be to allow more than one element at each node (not just at
the leaves). We would need an additional holder node that we could fill up to the capacity first, once filled we
would then insert it into our bag – basically we could use this holder to have each node in the bag hold the
same number of elements. I didn’t think of this solution right away and didn’t have time to implement it yet
(will do that for the final project).

4 Parallel BFS

The parallelization of BFS is based on the divide-and-conquer approach of splitting the nodes at layer \( k \) in
the graph (i.e. all the nodes a distance \( k \) away from the source) among the processors and assembling the
results at the end (this is done using a bag reducer). Partial code for the parallel BFS algorithm is presented
in Algorithm 1. There is a benign race condition in the code: when more than one node at level \( k \) point to
the same node \( u \) at level \( k+1 \), each processor working on the node at level \( k \) will add \( u \) to its output and we
might end up with more than one copy of a node in our output, which can slow down our computation. We
Algorithm 1 \textit{BFS\_PARALLEL}\hfill \\

\begin{algorithm}
\begin{algorithmic}
\STATE \textbf{Algorithm 1 BFS\_PARALLEL}
\STATE \textbf{BFS\_PARALLEL(s, distances)}
\COMMENT{Bag_redcuer curr\_level = new Bag_redcuer;}
\COMMENT{Bag_redcuer next\_level = new Bag_redcuer;}
\FOR {\textbf{i} = 0; \textbf{i} < nNodes; \textbf{i}++}
\COMMENT{distances[i] = INFINITY;}
\ENDFOR
\COMMENT{distances[s] = 0;}
\STATE \textbf{// while there are any items left in any of the 2 bags}
\STATE \textbf{// walk the bag for level k, insert elements into bag for level k+1}
\STATE \textbf{curr\_level->insert(s);}
\STATE \textbf{while}{(1)}
\STATE \textbf{BFS\_WALK\_LAYER(curr\_level->value, next\_level, distances);}
\STATE \textbf{if}{(next\_level->isEmtpy())}
\STATE \textbf{break;}
\STATE \textbf{swap(curr\_level, next\_level);}
\STATE \textbf{-------------}
\STATE \textbf{BFS\_WALK\_LAYER(Bag inB, Bag_redcuer outB, distances)}
\STATE \textbf{if}{(!inB->canSplit())}
\STATE \textbf{BFS\_WALK\_LAYER\_BASECASE(inB, outB, distances);}
\STATE \textbf{Bag* new\_bag;}
\STATE \textbf{// split this bag into 2 smaller bags}
\STATE \textbf{new\_bag = inB->split();}
\STATE \textbf{cilk\_spawn BFS\_WALK\_LAYER(new\_bag, outB, distances);}
\STATE \textbf{BFS\_WALK\_LAYER(inB, outB, distances, nodes, edges);}
\STATE \textbf{cilk\_sync;}
\STATE \textbf{}}
\end{algorithmic}
\end{algorithm}

could solve this by associating a bit with each node and use locking to update it (when a thread accesses the node), even without locking, the bit might help reduce the number of copies.

5 Results

The best results were achieved for the coarsened pennant bag. The leaf capacity that seemed to work well was 50 (the minimum required size for splitting was also set to 50).

Running the parallel BFS algorithm with Cilkview reports a parallelism of 193.44. However, we don’t see a significant speedup (Figure 2a). To investigate if this result is due to insufficient memory bandwidth, we can introduce more artificial work at each node of the graph (by making a call to the external library function trivial()). Indeed, as we increase the amount of work at each node, the speedup gets closer to being linear. Figure 2b shows the results when 1000 calls to trivial() were made at each node.
Lab 4: Parallel Breadth First Search

Ruben Perez

March 17, 2010

1 Introduction

This lab explores parallel breadth first search. Breadth first search is a graph search algorithm commonly used for single source shortest paths. It works by taking a source node, and examining all of its neighbors, nodes that share an edge with it. This set of nodes forms a frontier, a set of unexplored nodes, which are equidistant from the source. Note that this assumes that all edges are equal weight. For each node in the frontier, the process is repeated, with all discovered nodes added to a common frontier. This essentially causes all nodes that are distance 1 to be explored, then all nodes that are distance 2, and so on. A simple implementation for serial computation is to use a FIFO queue in order to maintain the frontier. Although the queue would actually contain both the frontier and all nodes that are 1 unit further from the source, the frontier’s frontier, the algorithm still processes the nodes in the correct order. One can see this by examining the nodes added to the queue. Since the queue is first in first out, one can see that as we iterate through a source’s neighbors, they are all added to the end of the queue and are distance d away from the source. By the time we pop off a node from this frontier, all nodes discovered will be distance d+1, since the nodes discovered are one distance away from a node who is d away from the source. However, no node ever discovered at this point will be less than d+1 away, since the next nodes in queue will all be at least d away. Obviously a queue does not easily lend itself to parallelism, so we will need a different data structure. The key observation is that given all nodes in a frontier, it does not matter in which order we process them. Thus a structure without order but supporting fast insertion and retrieval could work well in this situation.

2 Parallel Breadth First Search Data Structures

The data structure needed for the implementation of a parallel breadth first search is the bag. The bag is a data structure that contains elements in no particular order, and supports fast retrieval, insertion and union operations. By keeping the elements in a frontier in a bag, we can get away from the bottleneck of a queue. However, to implement a bag, we need another data structure
called a pennant. A pennant is a full binary tree of size \(2^r\) where \(r\) is some constant. However, its root has only child while every other node has two. This is important for operations on the data structure. A node in the pennant tree contains two pointers to children nodes, and a data element. A bag is represented by an array of size \(k\), where array\([i]\) is a pennant of size \(2^i\). A bag must be able to support an insert, and union operation as well as split operation. Consequently, the pennant structure must also support these operations.

### 3 Data Structure Implementation

Working from the lowest level up, a pNode consists of simply a piece of data, and pointers to two other nodes. It also supports an operation walk, which takes an vector as an argument. This operation adds the nodes data to the vector, and then calls walk on its children in order to get all data into the vector.

A pennant is a tree consisting of these nodes. The root has only child, which we represent as a pNode which has only a left child. The two important operations we consider are merge and split. Insertion can be considered as a merge of a single element pennant. Merge works by

1. Add a second root to pennant \(A\) and set it to be the child of pennant \(B\)'s root
2. Set the root of \(B\) to have the root of \(A\) as its only child
3. Return pennant \(B\)

Split essentially undoes these operations.

The bag is simply an array, \(A\), of these pennants where \(A[i]\) is a pennant of size \(2^i\). The size of this array dictates how large our bag is. To insert an element \(T\), we create a pennant with \(T\) as the root and then insert it as follows:

```cpp
Bag<T, BAG_PARALLEL>::insert(T el) {
    int k = 0;
    Pennant * x = new Pennant(el);
    while(bag[k] != NULL){
        x = this->bag[k]->merge(x);
        this->bag[k] = NULL;
        k++;
    }
    this->bag[k] = x;
}
```

As we iterate from \(k = 0\) onwards, we union the pennants and set array\([k]\) to be null. Since both pennants are full binary trees of size of \(2^i\), this is somewhat analogous to incrementing a binary counter as we carry numbers forward in the addition. The merge operation works in a way similar to a full adder where we have a carry pennant, and the two pennants, \(A\) and \(B\), being unioned. Thus we can sketch out the bag merge operation as:

```cpp
merge(A,B) {
```
carryPenant = null;
for k = 0 to MAX_ARRAY_SIZE
    (A[k], carryPenant) = FA(carryPenant, A[k], B[k])
}

The FA function is simply the full adder function between the three arguments which returns a tuple. The first element is the sum bit, and the second is the carry bit. The last operation needed is split, which can be thought of as a right shift.

Bag<T, BAG_PARALLEL>::split() {
    Bag<T, BAG_PARALLEL> *newBag = new Bag<T,BAG_PARALLEL>();
    Pennant *y = this->bag[0];
    this->bag[0]= NULL;
    for(int k = 1; k < 64; k++){
        if (this->bag[k] != NULL){
            newBag->bag[k-1] = this->bag[k]->split();
            this->bag[k-1] = this->bag[k];
            this->bag[k] = NULL;
        }
    }
    if(y != NULL){
        this->insert(y->root->data);
    }
    return newBag;
}

The note to make is that newBag represents the pennant that might get shifted off, the pennant of size one. By storing it, and setting that position in the pennant array, bag, to null, we essentially remove it from the bag. After shifting appropriately, we add it back to the bag. With these operations, we have a complete bag data structure ready for use in BFS.

4 Modifications to BFS

In contrast to the BFS that uses a queue to hold both the set of nodes that are d away from the source and the set that is d+1 away, we use bags to hold each set independently. We then take elements from the first bag, and examine them in parallel, adding the discovered nodes to the latter bag. After finishing every element in the first bag, we then create another bag and repeat the process. To support parallel operations, this data structure is wrapped in a reducer hyperobject, which prevents concurrency issues such as simultaneous merging and insertion.

int Graph::bfs_parallel(const int s, unsigned int *distances) {
    

TODO: Fill me in

fprintf(stderr, "Running: bfs_parallel\n");
unsigned int newdist;

Bag_red<inf, BAG_PARALLEL>::Bag_red * layer_k = new Bag_red<int, BAG_PARALLEL>();
if (s < 0 || s > (int)nNodes)
    return -1;
//set all distances to infinity
cilk_for (unsigned int i = 0; i < nNodes; i++) {
    distances[i] = D_INFINITY;
}
//initialize bag and starting distances
distances[s] = 0;
newdist = 0;
layer_k->insert(s);

while ( (!layer_k->is_empty()) ) {
    Bag_red<int, BAG_PARALLEL>::Bag_red * layer_k_plus = new Bag_red<int, BAG_PARALLEL>();
bfs_walk_layer<BAG_PARALLEL>( layer_k, layer_k_plus, newdist, nodes, edges, distances);
    layer_k = layer_k_plus;
    newdist++;
}
return 0;
}

The bfs_walk_layer function uses a divide and conquer strategy to split the bag representing nodes that are d away stored in a bag reducer layer_k into manageable chunks of size NEIGHBOR_CHUNK. The observation to make is that all these chunks can be operated on in parallel. At this point we call a function bfs_walk_layer_base which examines all nodes in the bag, and adds neighbors to the bag layer_k_plus, which represents nodes that are d+1 away. A template implementation of these functions is:

template <BagType BType>
static int
bfs_walk_layer(Bag_red<int, BType> *in_bag, Bag_red<int, BType> *out_bag, unsigned if(in_bag->get_reference().num_elements() < NEIGHBOR_CHUNK){
    bfs_walk_layer_base(in_bag, out_bag, d, nodes, edges, distances);
    return 0;
}

//inefficient way of getting all elements in bag while an iterator is developed
Bag<int,BType> * new_bag_bag = in_bag->split();
typename Bag_red<int, BType>::Bag_red * new_bag = new Bag_red<int, BType>(
    std::vector<int> * new_bag_vec = new_bag_bag->copy_to_vector();
typedef std::vector<int>::iterator it;
for(it = new_bag_vec->begin(); it < new_bag_vec->end(); it++) {
    new_bag->insert(*it);
}
cilk_spawn bfs_walk_layer<BType>(new_bag, out_bag, d, nodes, edges, distances);
bfs_walk_layer<BType>(in_bag, out_bag, d, nodes, edges, distances);
cilk_sync;
return 0;
}

template <BagType BType>
static int
bfs_walk_layer_base(Bag_reducer<int, BType> *in_bag, Bag_reducer<int, BType> *out_bag, unsigned

typename std::vector<int>::iterator it;
std::vector<int> * in_bag_vec = in_bag->get_reference().copy_to_vector();

for (it = in_bag_vec->begin(); it != in_bag_vec->end(); ++it) {
    for (int i = nodes[*it]; i < nodes[(*it) + 1]; i++) {
        if (distances[edges[i]] == D_INFINITY) {
            distances[edges[i]] = d + 1;
            out_bag->insert(edges[i]);
        }
    }
}
free(in_bag_vec);
return 0;
}

5 Results
A good deal of time was spent in the actual implementation of the bag structure, which did not leave much time for use in BFS. While the implementation was successful, as evidenced by passing the unit tests, as well as use in serial BFS, it was considerably slower than using queues, about a 4x slowdown. Of course, this is not unusual since the data structures for the bag are considerably more complex. Unfortunately, when used in in the parallel version, most nodes were never visited, as their distance to the source remained infinity and other nodes had distance that were much larger than they should have been. This indicates a data race of some sorts, particularly as the serial version using the same data structures works (setting cilk worker count to 1). There is a benign data race in the setting of the distance in the base case of walking a layer. However, since the value would be set only if the node had not yet been discovered (its distance
is still infinity). Even if the node was discovered simultaneously, because we are working on nodes that are all equidistant, the distance should be update to the same value, which doesn’t explain overly large values for distances to nearby nodes. Interestingly enough, cilkscreen does not show either of these errors. Unfortunately, without resolving the first issue, further analysis isn’t possible.

6 Further Work

The paper on PBFS by Leiserson and Schardl described an effective coarsening method which consisted of using an array to hold elements, and using these arrays as the data of nodes, rather than a single element. I had also planned to explore the 0-1 edge problem, where edges in the graph are of weight one or zero. I hadn’t given a great deal of thought, but it seems that it may be possible to add nodes that are discovered from 0 edges back into the input bag, or a subdivision thereof, seeing as they are the same distance away from the source as the nodes in the input bag. This could generate a lot of extra work, since it’s conceivable that nodes get added multiple times to bags, particularly if there is a path of nodes where all edges are 0. In this case all nodes in this path will be added, but they will be added again if another node is connected to the same path. A bit map could be used to check if a node has already been added. This introduces a data race, but like updating distances, the worst case will be that the node is added multiple times.
For this lab, implementing parallelized BFS, due to difficulties debugging I hit the 12-hour limit well before being finished. I completed the part of the lab I was on at the time, going a couple hours past that time limit, but still only having done the list-based bag implementation. A partial writeup follows.

1 BFS Layering

A simple serial BFS algorithm will never contain nodes in the queue from multiple layers. To prove this, we assume unweighted graph edges. Given that we start with some node \( s \) and calculate distance as distance from that \( s \), then the algorithm will add all nodes at distance 1 to the queue first. Only then will it add nodes adjacent to those distance 1 nodes, after \( s \) which is all distance 0 nodes, has been finished and is no longer in the queue. For any distance \( n \), distance \( n + 1 \) will currently be added to the queue, with all nodes at distance \( n + 1 \) being added. Any nodes with distance less than \( n \) will already have been visited and thus not added to the queue, while any nodes with distance greater than \( n + 1 \) will not be adjacent to the distance \( n \) nodes since edges are unweighted.

Implementing a two-queue BFS algorithms increased running times on the sample datasets by about 5%.

2 BFS with Bags

Bags actually seem like a great idea. I had a lot of difficulty implementing a search with the basic list-based Bag. Sadly, this was due to bad debugging of problems caused by poor code assumptions rather than anything else. This implementation actually performed a bit slower than the serial version, taking 20% to 40% more time depending on the dataset.

There is a race based on multiple threads working on nodes which connect to the same adjacent node as each other at the same time. In this case both might try to modify that adjacent node’s distance and add it to the next level’s bag at the same time. This is a benign race, though, as the duplicate distance setting will be set to the same value for both and processing the node twice in the next level will have the same effect as any other situation where all adjacent nodes have already been visited.

This does seem to have limited scalability due to the split operation. Because it splits a fixed amount rather than splitting in half, we can only reasonably split each bag up into two portions. The parallel bag structure which I was unable to get to looks to solve this.
Lab 4 – Kevin Kelley

(a) The code provided builds and executes correctly.

(b) It is clear that, in the BFS algorithm provided in Fig. 1, the queue never contains nodes from more than two distinct layers. This follows from two observations. First, the item at the front of the queue will always be of minimal layer. Second, from a node in layer $i$ we can only reach nodes in layer $i + 1$. Thus, while we are processing nodes in layer $i$, there may be some nodes from the next layer in the queue (as they are generated) but we cannot begin adding nodes in layer $i + 2$ to the queue until we dequeue a node in layer $i + 1$, at which time there cannot be any nodes in layer $i$ still on the queue. The implementation bears out this conclusion and performs identically to bfs_serial. However, I got a small but significant improvement (0.254 sec from 0.367 sec) by removing the inline conditional, which is entirely unnecessary.

(c) The linked-list implementation does, in fact, compile and pass the test suite.

(d,e) The serial bag-based implementation takes 0.449 sec. The overhead from splitting and merging (necessary to expose parallelism) is absolutely intense. Even the serial implementation shows a slowdown by a factor of 136.

<table>
<thead>
<tr>
<th>Workers</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>61.257 sec</td>
</tr>
<tr>
<td>2</td>
<td>65.385 sec</td>
</tr>
<tr>
<td>3</td>
<td>67.924 sec</td>
</tr>
<tr>
<td>4</td>
<td>68.262 sec</td>
</tr>
<tr>
<td>5</td>
<td>69.635 sec</td>
</tr>
<tr>
<td>6</td>
<td>75.775 sec</td>
</tr>
<tr>
<td>7</td>
<td>77.755 sec</td>
</tr>
<tr>
<td>8</td>
<td>76.206 sec</td>
</tr>
</tbody>
</table>

As the assignment suggests, there is in fact a data race, but it is benign; the race is on distances, but since each vertex can only appear once per layer, we won’t ever have two workers writing to the same place.

Not only is this vastly slower than the serial implementation; the extreme contention between threads means that there is a general trend of slower performance as the number of workers increases. Needless to say, this is not the sort of result in which we are generally interested.

(f) The pennant implementation was relatively straightforward; I also constructed a simple test suite that verifies that union and split operations are correct. Each pennant consists of a group of nodes with a parent pointer and child pointers. I also implemented shallow and deep copy functions.

I implemented the bag on top of this as described in the assignment (on pages 4 and 5); each slot in the bag contains a pennant with size double the previous slot. I used a fixed-size, 32-slot array for my bags; this allows for $2^3 3 - 1$ elements in a bag, which is plenty for our needs.
I replaced the linked-list bag implementation with the pennant-based one. To my surprise, even after I spent some time improving the quality of my implementation, I barely saw any performance improvement (about a factor of 1.5) over the linked-list implementation.

<table>
<thead>
<tr>
<th>Workers</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48.421 sec</td>
</tr>
<tr>
<td>2</td>
<td>41.951 sec</td>
</tr>
<tr>
<td>3</td>
<td>38.054 sec</td>
</tr>
<tr>
<td>4</td>
<td>37.643 sec</td>
</tr>
<tr>
<td>5</td>
<td>36.963 sec</td>
</tr>
<tr>
<td>6</td>
<td>36.851 sec</td>
</tr>
<tr>
<td>7</td>
<td>36.753 sec</td>
</tr>
<tr>
<td>8</td>
<td>36.635 sec</td>
</tr>
</tbody>
</table>

I coarsened the leaves of the pennant by implementing a “hopper.” This is effectively a small, fixed-size array that is filled as elements are inserted into the bag. When the array fills, it is inserted into the bag. We expect this to provide an enormous improvement, since only one in $k$ insertions is anything but a simple constant-time array insertion; and it does. However, we still see the same limited parallel speedup. I settled on $k = 50$, although this might have been tuned further. This produced an approximately constant-factor speedup of about 35.

<table>
<thead>
<tr>
<th>Workers</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.353 sec</td>
</tr>
<tr>
<td>2</td>
<td>1.216 sec</td>
</tr>
<tr>
<td>3</td>
<td>1.105 sec</td>
</tr>
<tr>
<td>4</td>
<td>1.088 sec</td>
</tr>
<tr>
<td>5</td>
<td>1.062 sec</td>
</tr>
<tr>
<td>6</td>
<td>1.043 sec</td>
</tr>
<tr>
<td>7</td>
<td>1.029 sec</td>
</tr>
<tr>
<td>8</td>
<td>0.994 sec</td>
</tr>
</tbody>
</table>

As the lab suggests, I checked with CilkView that sufficient parallelism was exposed. The numbers reported (approximately 15) should have been more than sufficient for eight processors. The experimentation suggested seems to confirm that this is an issue related to memory bandwidth.

Had I had more time, I would have liked to experiment with modifying data structures in an attempt to make them smaller. I’m not sure exactly how I would have done this, but perhaps it could have improved the situation.
Deleting from bags

Swap-and-delete

When a 1-element pennant exists (in the lowest-order slot of the bag) we can simply remove it (deallocating it and replacing it with a NULL pointer) and have a correct bag. If this pennant’s single element were to be the one we were attempting to delete, we would be all set; therefore, we can simply search for that element and swap it with the element in the 1-element pennant (since order is irrelevant).

Obviously this requires searching for the element to be deleted. In this implementation I avoided this by returning a handle from the insertion operation that allowed $O(1)$ location of the element to be removed. In a real implementation, to avoid a linear search, an auxiliary data structure might be necessary.

Deleting by splitting

When a 1-element pennant does not exist, we can create one by more or less reversing the process presented in the assignment. At the end, we will be left with two 1-element pennants; one is placed in the appropriate slot in the bag, and the other is used for the swap-and-delete described previously.

This is approximately as fast as the pennant-based bag insertion, before any optimizations were applied; that is to say, it is rather slow. Performance considerations resulting from this basic algorithm for deletion are described in more detail below.

Removing from “hopper” first

As described previously, I use a “hopper” arrangement to coarsen the leaves of the pennants. Obviously, if the hopper is not empty, we can simply swap the element to be deleted into the hopper (unless it is already there) and then delete it. This takes $O(1)$ time, and has precisely as dramatic an effect on deletions as the coarsening had on insertions.

This makes sense, since in both cases, a hopper size of $k$ means that we only have to actually perform the operation in question once every $k$ operations. A test case that interleaved insertions and deletions ceased to be particularly useful, since this effectively boils down to a series of cheap constant-time operations; if the hopper is never filled or emptied, no actual insertions or deletions from the bag must be performed.

When the hopper does empty, the procedure above will not work, and we must delete from the bag as described in the first section. The removed second 1-element pennant is placed in the empty hopper, and then the deletion occurs.
Remaining performance issues

There is a fundamental problem with performing insertions and deletions in the pennant-based bag structures. It is easiest to explain by analogy to binary arithmetic. Imagine that a 1 represents the presence of a pennant and a 0 represents its absence. Insertions and deletions are then analogous to adding or subtracting one.

The problem is that we may have a chain-carry or chain-borrow in our arithmetic. This requires a series of operations. Since each is constant-time, this is perhaps not terrible. However, consider the example from the previous section in which we have interleaved insertions and deletions. In that context, these were advantageous. However, what if those operations take us back and forth across one of these boundaries? This results in a performance problem.

Faking deletes with placeholders

To reduce the effects of this problem, I considered avoiding actual deletions by replacing deleted elements with a “no element” placeholder when there was not an element in the hopper to swap. This would allow overlapping boundaries for actual insertions and deletions, much the way they are employed for splits and merges in traditional graph algorithms.

I cannot think of a reason why this should not correct the worst-case issue described, but I did not have a chance to implement or test it to determine whether it had any unexpected performance effects.