Final Project Report

Available online on Google Docs.

Andres Riofrio — CS 140

I wrote and tested three different implementations of a distributed queue for Breadth First Search on *directed* graphs. The first one is a linked list structure as presented in this MIT handout. The second one is a "smart" linked list structure, which does not split the queue all at once but amortizes the cost of splitting among the workers by skipping items when traversing the queue. The third one is a pennant bag structure, as described by Leiserson (1). A bag has a "backbone" of pointers that are either null or point to a pennant that has exactly 2^k elements, where k is the zero-based index of the pointer in the backbone. This structure behaves very much like a binary number, achieving O(log max(n)) worst-case for insertion, split, and union. Finally, I implemented a Graph 500-compatible verifier.

Implementation Notes

In my implementation, there is a benign race condition in which two workers can try to add the same node to their own queue (which later get merged) by first checking whether any worker has already added that node. It is thus possible for the same node to be added multiple times to the queue by multiple workers. When this happens, some additional work is created, since the node will be processed multiple times on the next round. This means multiple workers will attempt to add the node's children to the queue. One can imagine how this could exponentially increase the amount of work. However, since the workers will first check whether or not the child has been added to the queue by any worker, it is unlikely that the children will be added multiple times to the queue.

If a node was added multiple times to the queue, another benign race condition occurs when updating the results (level and parent node) for the node. Thankfully, since all workers are on the same round, they will agree on the level, and the results are valid no matter which parent node is chosen.

Challenges

The original implementations kept track of which nodes had been added to a queue through a C++ `deque<bool>`. A `vector<bool>` was not usable because it is specialized for compactness, so that many items are stored in the same word, making it unsafe to access from multiple workers at the same time. However, the `deque<bool>` introduced considerable overhead, so much that it overshadowed any performance difference between the three algorithms tested.

This strategy was replaced with the strategy used by Leiserson. Instead of checking a dedicated array of booleans, we can check if `node_level` has been changed (from the default of -1). If it has, then we know the node has already been added to a queue. We also move the job of updating the results (`node_level` and `node_parent`) to right before the node is added to the queue, instead of after it is read off from the queue (in the next round).

Another change that made a big impact in the numbers was adding `-O3` to the compiler options. This inlined a lot of function calls and cut execution time by 4.

Algorithms Tested

I used the same Breadth-First Search algorithm described by Leiserson, and tested three different distributed queues. Grainsize was min(2048, n / 8p), where *p* is the number of workers used and *n* is the number of nodes in the queue at each level. The basic algorithm is as follows:

```
void BFS::run() {
  bag<int> queue, next;
  queue.insert(root);
  node_level[root] = 0;
  for(int level=0; !queue.empty(); level++) {
    process_queue(queue, next, level);
    queue.swap(next);
    next.clear();
  }
}

void BFS::process_queue(bag<int> &queue, bag<int> &next, int level) {
  if(queue.size() <= grainsize) {
    for(bag<int>::iterator it = queue.begin(); it != queue.end(); it++) {
      int node = *it;
      for(int i=0; i<graph.degree(node); i++) {
        int neighbor = graph.neighbor(node, i);
        if(node_level[neighbor] == -1) {
          node_level[neighbor] = level + 1;
          node_parent[neighbor] = node;
          next.insert(neighbor);
        }
      }
    }
  } else {
    bag<int> right_queue;
    queue.split(right_queue);

    bag<int> right_next;
    cilk_spawn process_queue(queue, next, level);
    process_queue(right_queue, right_next, level);
    cilk_sync;

    next.merge(right_next);
  }
}
```

Naïve linked lists

The strategy is to store nodes in different linked list for each Cilk++ strand, and join the left and right linked lists on each knot. The weak link is the split operation, which must traverse through half of the list. To calculate the cost of this strategy, let n be the number of nodes in this level and n' be the number of nodes in the next level. Then, the total cost is

$$O(\log p + n'/p + \log p + n \log p + n/p) = O(n \log p + \log p + n'/p + n/p)$$

Create: O(1) executed log(n/g) = O(log p) times

Create a linked list.

Insert: O(1) executed O(g · n'/n) = O(n'/p) times

Insert an element to the end of the linked list. Also updates an internal size counter.

Union: O(1) executed O(log p) times

Set the "next" pointer of the last element of the first list to point to the first element on the second list. Implemented using `list::splice()`.

Split: O(n) executed O(log p) times

Walk through the list $n/2$ times, where n is the number of elements in the list (kept track of as an internal counter updated every time an element is inserted). Then, set the "next" pointer of the last element to none, and set the "first" pointer of a new list to the value of the pointer that was just overwritten.

Next: O(1) executed g = O(n/p) times

Following a pointer to reach the next item is easy.

Smart linked lists

Attempting to improve on the previous model, this strategy eliminates the cost of splitting the list, at the cost of some overhead when walking the list. The total cost of this strategy is:

$$O(\log p + n'/p + \log p + \log p + pn/p) = O(\log p + n'/p + n).$$

Compared to the other strategy, we replace $O(n \log p + n'/p)$ with $O(n)$.

Create: O(1) executed O(log p) times	Same as above.
Insert: O(1) executed O(n'/p) times	Same as above.
Union: O(1) executed O(log p) times	Same as above.
Split: O(1) executed O(log p) times	This is a no-op.

Next: O(p) executed O(n/p) times

Skip $s - 1$ items on the queue (these items will be processed by another strand). The number of strands $s \approx n/g = O(p)$. Note that at the beginning of each strand, an operation of up to O(s) is executed to skip items that will be processed by another strand.

Pennant bags

A bag is a set of $n = a_0 + 2a_1 + 4a_2 + 8a_3 + \ldots + 2^r a_r$ elements. These elements are stored in r + 1 pennants, each of which contain exactly 2^k elements. A pennant is a binary tree node with no right child and a left subtree that is perfect and complete (or empty). As described by Leiserson, operations on a pennant bag are very much like operations on binary numbers. Namely, insertion is like the increment operator, union is like ripple-carry addition, and splitting is like bit-shifting. Finally, the bag has a capacity 2r that it must not exceed. In our case, $2r = E$, where E is the total number of edges in the graph. The total cost of this strategy is:

$$O(\log E \log p + n'/p + \log E \log p + \log E \log p + n/p) = O(\log E \log p + n'/p + n/p)$$

Compared to naive linked lists, we replace O(n log p + log p) with O(log E log p). Compared to smart linked lists, we replace O(log p + n) with O(log E log p + n/p).

Create: O(log E) executed O(log p) times

Allocate an array of size log E to hold the backbone, and reset each item to null.

Insert: O(1) amortized time executed O(n'/p) times

Behaves like a binary counter. Also updates an internal size counter.

Union: O(log E) executed O(log p) times

Behaves like a binary ripple-carry adder. Also updates both internal size counters.

Split: O(log E) executed O(log p) times

Behaves like binary bit shift. Also updates both internal size counters.

Next: O(1) executed O(n/p) times

Walk through a pennant (a binary tree). If we are done with the current pennant, start with the next non-null pennant.

Validation

Every time a BFS search is run, the following validation is done (from the Graph 500 specification):

1. the BFS tree is a tree and does not contain cycles,
2. each tree edge connects vertices whose BFS levels differ by exactly one,
3. every edge in the input list has vertices with levels that differ by at most one or that *the parent (the first in the edge tuple) is not in the BFS tree* (the Graph 500 specification says "or that both are not in the BFS," which is inappropriate for directed graphs),
4. the BFS tree spans an entire connected component's vertices, and

5. a node and its parent are joined by an edge of the original graph.

See the `BFS::validate()` method in the code. When measuring the performance of each algorithm, I did not include the time to parse the graph or construct it, nor the time to verify the results.

Results

I ran three different implementations of each algorithm: one that used deques (as described in the Challenges section), another without deques, and another without deques and compiled with `-O3` optimization on. Each of these implementations was an improvement over the previous one, but the difference between the algorithms remained negligible, especially in terms of scalability. The conclusions above about how the different algorithms would affect execution time are largely irrelevant under the conditions tested, because the common algorithm takes over most of the time. I present a set of results I gathered with the first implementation (using deques) and then two tests I was able to gather with `-O3` optimization on. All the data is available online as a spreadsheet on Google Docs.

Scalability across number of edges reached

We can see in these graphs that execution time is, at least in the deque implementation, directly proportional to the number of edges reached, approximated by multiplying the number of nodes reached by the average node degree. We also see that both the naive and smart linked list algorithms behave similarly (I was not able to test pennant bags under this configuration). Finally, we see that the number of workers affects the slope of the line. If we assume perfect efficiency, we get that execution time is O(pn). However, we will see in the following experiments that neither algorithm has perfect efficiency.

(See graphs on page 7.)

First test: Small number of nodes reached, large average degree

I ran a test with 2^{19} total nodes and a total edge factor of 300. Among the 6095 nodes reached by the BFS algorithm, the average degree of nodes was 25,805.80. There were 4 levels reached, including the root.

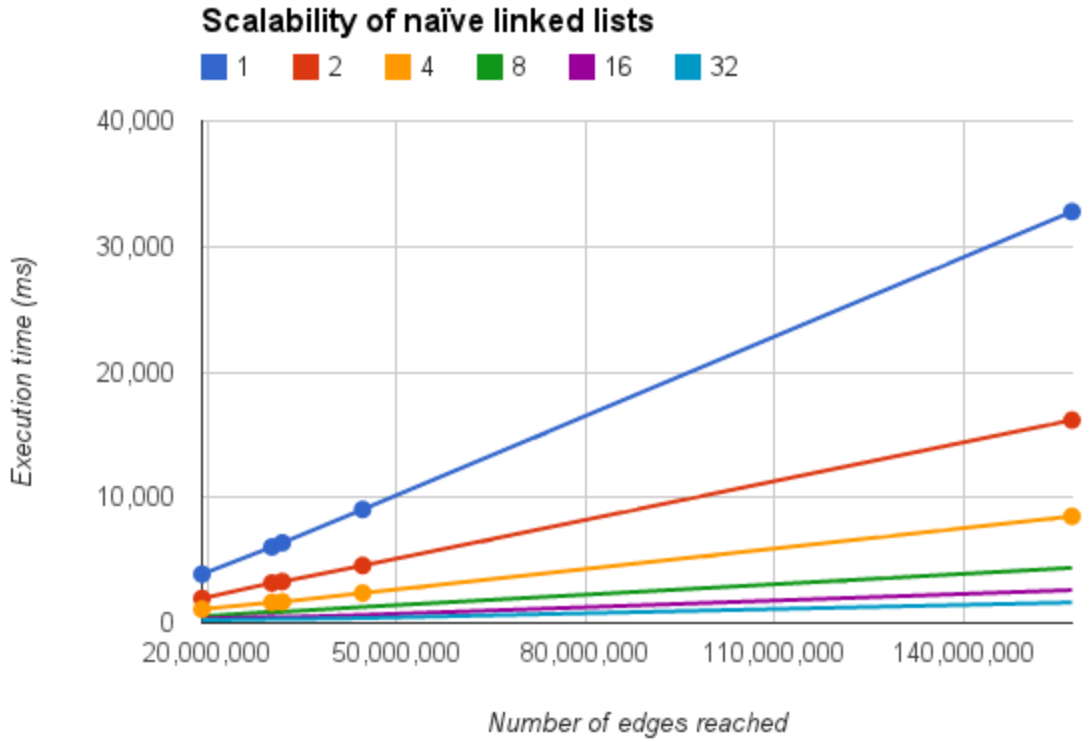
Even though the pennant bag implementation is up to 1.5 times faster than the other implementations, it seems to scale across number of workers worse than them.

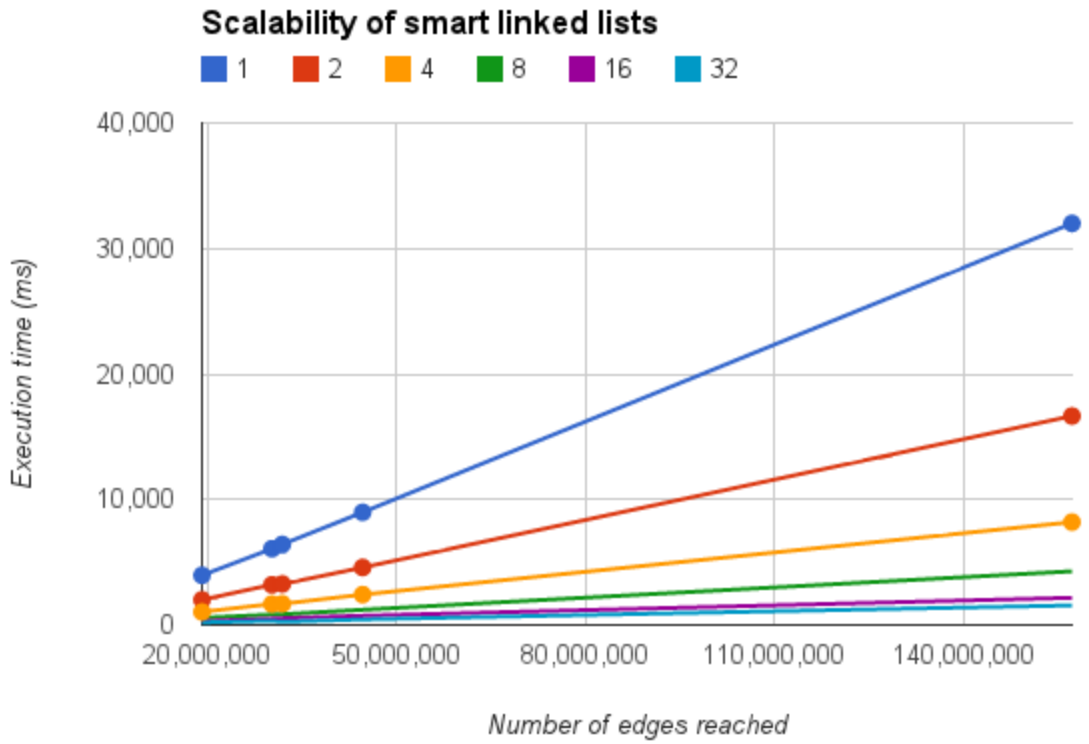
(See graphs on page 8.)

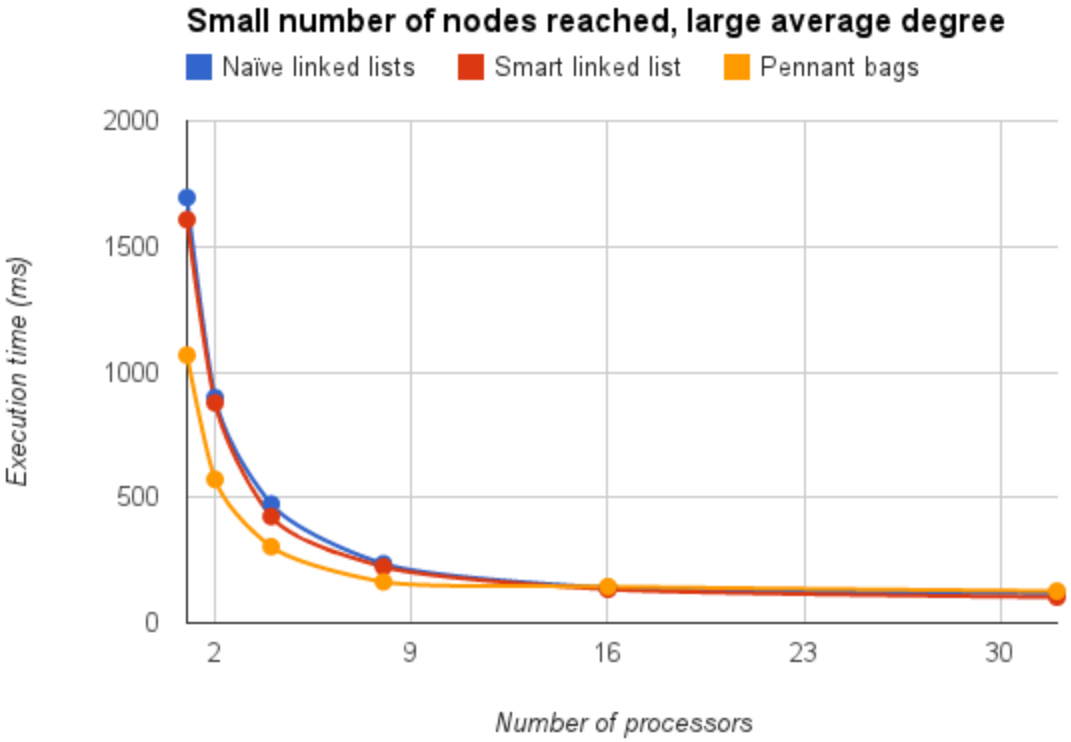
Second test: Larger number of nodes reached, small average degree

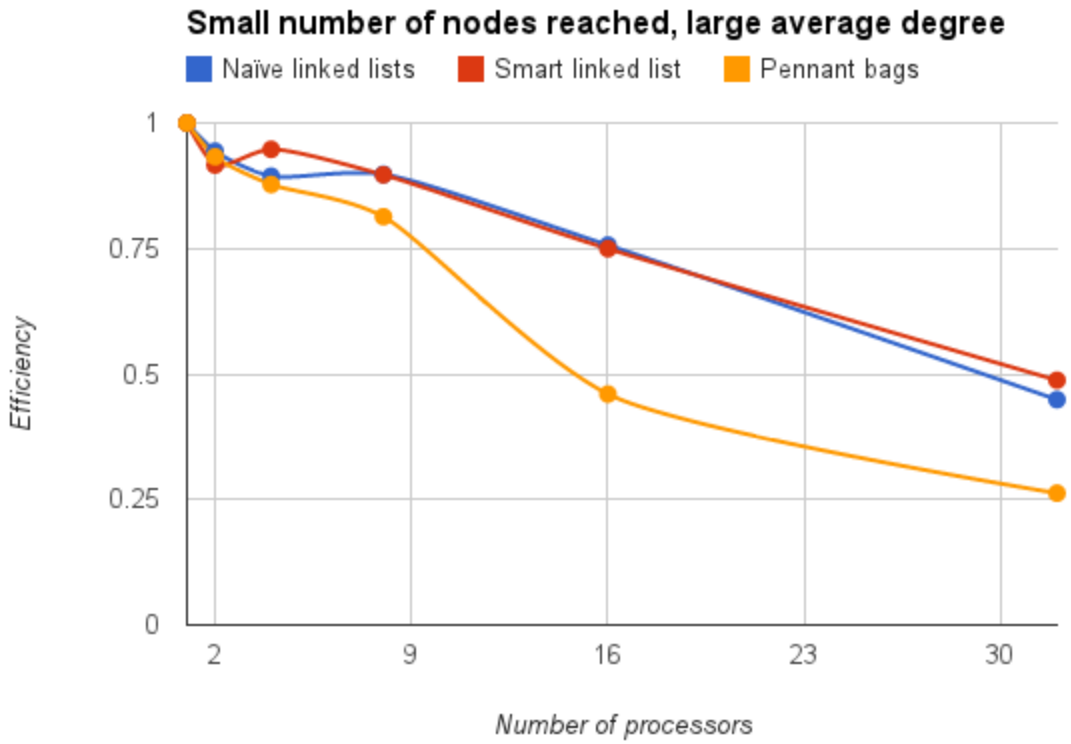
Another test was ran with 2^{17} total nodes and a total edge factor of 300. Among the 18,188 nodes reached by the BFS algorithm, the average degree of nodes was 1,658.93. There were 5 levels reached, including the root.

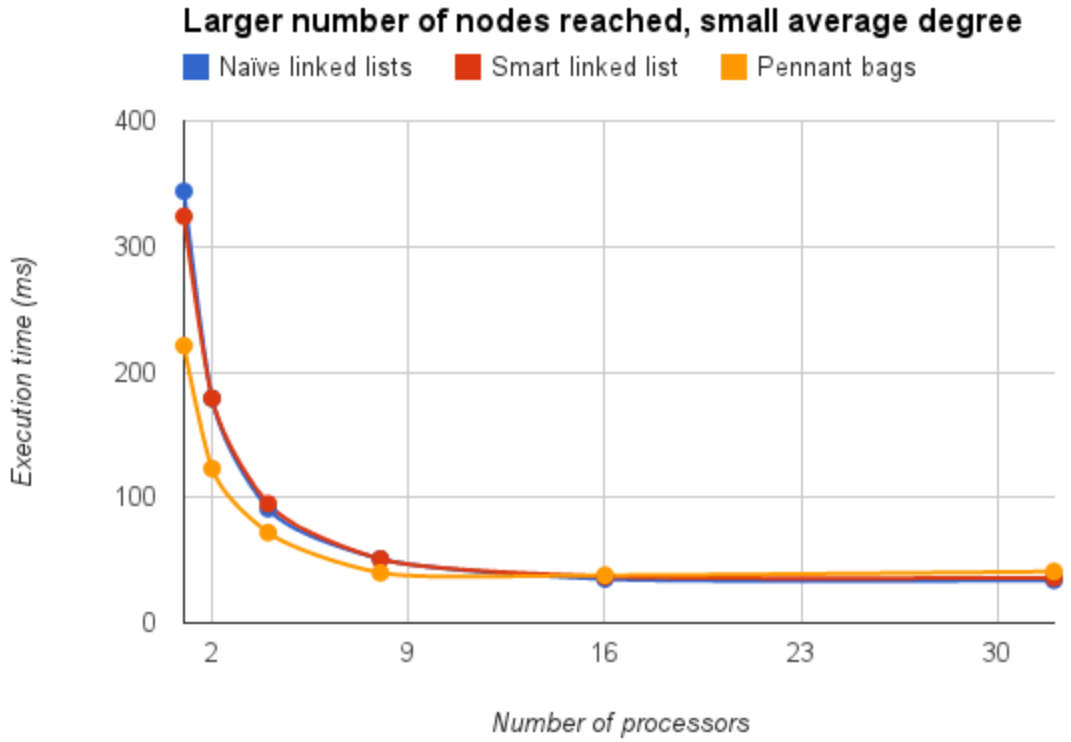
We see pretty much the same results as we saw on the first test. (See graphs on page 9.)

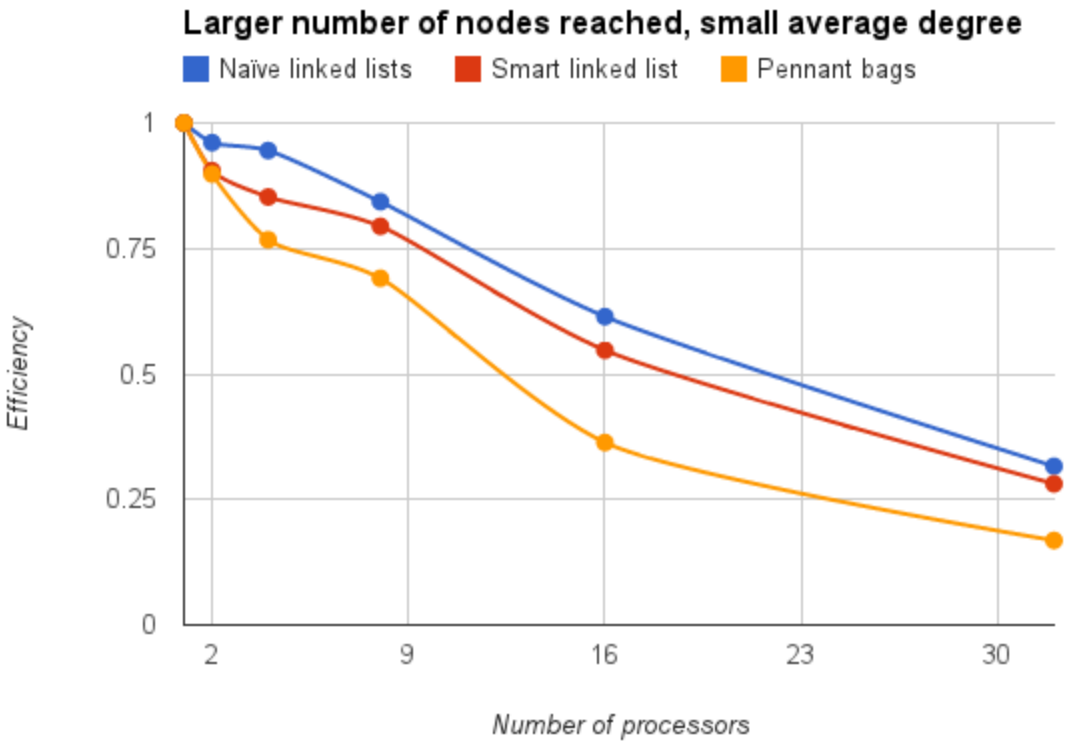

Scalability of naïve linked lists
1
2
4
8
16
32
40,000
30,000
20,000
10,000
0
Execution time (ms)
20,000,000
50,000,000
80,000,000
110,000,000
140,000,000
Number of edges reached


Scalability of smart linked lists
1
2
4
8
16
32
40,000
30,000
20,000
10,000
0
Execution time (ms)
20,000,000
50,000,000
80,000,000
110,000,000
140,000,000
Number of edges reached


Small number of nodes reached, large average degree
Naïve linked lists
Smart linked list
Pennant bags
2000
1500
1000
500
0
Execution time (ms)
2
9
16
23
30
Number of processors


Small number of nodes reached, large average degree
Naïve linked lists
Smart linked list
Pennant bags
1
0.75
0.5
0.25
0
Efficiency
2
9
16
23
30
Number of processors


Larger number of nodes reached, small average degree
Naïve linked lists
Smart linked list
Pennant bags
400
300
200
100
0
Execution time (ms)
2
9
16
23
30
Number of processors


Larger number of nodes reached, small average degree
Naïve linked lists
Smart linked list
Pennant bags
1
0.75
0.5
0.25
0
Efficiency
2
9
16
23
30
Number of processors

References

1. Charles E. Leiserson, Tao B. Schardl, *A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers),* 2010. Link.