Cilk++ Breadth First Search: Final Report

Nicholas Vaughan Kevin Wojcik

March 22, 2012

1 Introduction

Our project is the implementation of a multi-threaded breadth first search on large undirected graphs for shared memory computers. In particular, it implements the search portion of the Graph500 benchmark.

The implementation uses an interesting data structure to store graph nodes in, which allows for very efficient operations.

While writing our project, we performed a number of experiments on various nodes of Triton which helped us learn about how our code performed on real-world machines. These experiments helped us to quantify improvements and make sure we were on the right track.

2 Breadth First Search and the Bag Interface

Before we detail our implementation of breadth first search, it is necessary to explain the `bag` interface, which consists of the following operations:

- Insert one element,
- Merge two `bag`s into one `bag`,
- Split a `bag` into two `bag`s of equal size, and
- Dump the elements of the `bag` to a `vector` (for iteration).

The `bag` interface does *not* require that all elements in a given `bag` be unique. In addition, `bag`, with merging two `bag`s as the binary operation, and the empty `bag` for the identity, forms a monoid with which we make a Cilk++ reducer. Our implementation is not designed to be re-used, and so the element type is fixed as `int`, for simplicity.

Leaving aside for a moment the implementation of `bag`, we can now talk at a high level about the implementation of breadth first search. Our breadth first search is composed of sequential rounds of searching, each one examining the next-nearest level. In each round, it starts off with the `bag` of nodes found in the previous round (or just the starting node, if this is the first iteration), and performs a very simple divide and conquer search.

First, the method asks the `bag` how big it is.

- If its size is larger than the grain size, split the `bag` and recurse twice (one of which on a different thread, with `cilk_spawn`).
- If its size is less than or equal to the grain size, iterate over the elements of the `bag`. For each node in the bag, iterate over its neighbors. For each of its neighbors, check whether it has been found.
  - If it has, continue on.
  - If it has not, mark it as found, update its level number and parent node, and add it to the next round's `bag`.
- If the bag is empty then there were no nodes found in the last round, and the search is complete.

When the threads terminate execution, the multiple views of the next round's `bag` will be merged together ('reduced', to use Cilk's terminology) in parallel.

3 Pennant-Based Bag Implementation

In this section, we discuss our particular implementations of these algorithms, together with an analysis of the running time of each operation. The information in this section is based heavily upon "Handout 10: Lab 4: Breadth First Search," from MIT, as reproduced on the Gauchospace page for this project.

3.1 Pennants

A pennant is a *complete* binary tree on $2^k - 1$ elements (for some natural number k), together with one additional element which is stored separately from the tree. Since the pennant is *complete*, it has a depth of k.

Pennants support the following operations:

- Create from 2^k elements,
- Merge with a pennant of equal size, to create a pennant of size 2^{k+1}, and
- Dump all elements of this pennant to a `vector`.

Once again, please note that elements stored in a pennant are not necessarily unique. Also note that the number of elements in some pennant is always some power of two.

3.2 Pennant Algorithms

- Create a pennant of size 2^k:

  Take one element and hold onto it outside of the tree. Then, take the other elements and build a complete binary tree out of them.

  Time: $O(n)$.

- Merge two pennants P and Q, both of size $n = 2^k$:

  Create a new, empty pennant, R. Set its single extra value to Q's single extra value. Set its tree to a new tree, which has P's tree as the right branch, Q's tree as the left branch, and P's extra value as its root.

  Time: $O(1)$.

- Dump all n elements:

  Walk the binary tree and push each element as you go. Then, push the extra value.

  Time: $O(n)$.

3.3 Bags

A `bag` consists of an array of pointers to pennants. Element i of the array must be either `NULL` or a pointer to a pennant of size 2^i.

It follows that a bag with n elements in it will have `bag`s in positions correspondent to the one bits in the binary representation of n. As we will show, operations on a pennant-based `bag` look similar to the analogous operations on bits, and have similar complexity.

3.4 Bag Algorithms

- Add a pennant of size $m = 2^k$ to this `bag` of size n:

  Check to see if there is already a pennant in index k of the array.

  - If there isn't, put this one there and stop.
  - Otherwise:
    1. Remove the pennant at index k from the array,
    2. Merge the two pennants
    3. Recursively insert the merged pennant.

  Since this method recurses at most $\log_2(n) + 1$ times, its running time is $O(\log(n))$. The analogous binary arithmetic method is addition of a power of two to a number.

- Insert one element to a `bag` containing n elements:

  Create a pennant with one element in it. Then, add this pennant to the `bag`.

  Time: $O(\log(n))$.

- Merge two `bag`s of size $\leq n$:

  For each pennant in the first `bag`, insert it into the second `bag`.

  Time: $O(\log(n)^2)$.

  Note that a more efficient implementation of this algorithm could perform a maximum of $\log(n)$ carries (by keeping track of them while advancing up the bits, rather than this more naïve implementation). This would give it a complexity of $O(\log(n))$. Fortunately, our implementation is much more straight-forward given the building blocks we already have in place. This only represents a minor short-coming of our code since this running time is dwarfed by everything else in the search.

  If we had gone with the more complex implementation, this would correspond to general binary addition.

- Split a `bag` of size n into two `bag`s of equal size:

  Create a new, empty `bag`. For each non-`NULL` element in the pennant array, aside from the zeroth element:

  1. Create two new pennants from the existing pennant of size 2^k.
  2. Set the $(k-1)^{\text{th}}$ element of the array of the `bag` to one of the the new pennants, for each `bag`.

  As both of these operations are constant time, and there are $\log(n)$ iterations, this takes $O(\log(n)$ time. The analogous binary arithmetic operation to this is division by two (more intuitively, right shift by one).

- Dump all n elements:

  For each pennant, ask it to dump its elements. Since there are at most $\log_2(n)$ pennants, the overhead is negligible.

  Time: $O(n)$.

4 Experiments

During the development of our project, we performed a number of experiments. The order they are presented in should not be mistaken for chronological order.

4.1 Augment "level" Array with Bitfield

In order to check whether a given node has been found yet, we index into an array of `int`s to determine its level. If its level is -1 (unset), then it is considered not-yet found.

The experiment performed was to add a bitfield with one bit per node denoting whether that node had been found yet or not, then change the search code to use it before the `level` array. Our hypothesis was that this would reduce the number of cache misses and speed up

the code, since one cache line of a bitfield contains many more bits than one cache line of an `int` array contains `int`s.

When we performed the experiment, however, we discovered that the version of the code using this bitfield is actually *slower* than the non-bitfield version. The original search code ran in 897ms on a representative RMAT graph (standard Graph500, but with 2^{23} nodes), and the revised code took 955ms. These results were consistent between runs.

One possible reason that this "optimization" turned out to be a flop is that most of the search's time is spent on the early rounds of the search, when most of the nodes are unexplored (and so almost all checks in the bitfield are almost immediately followed by setting an element of the `level` array, making the cache profile *worse* rather than better).

4.2 Replace Set with Bag

Before we decided that implementing a pennant-based `bag` would be a good idea, our code used a reducer which consisted of STL `set`, together with the union operation.

Unfortunately, this method had an extremely major pitfall that we had not initially realized. Since STL `set` does not have random access iterators, iterating through the new elements and giving them to threads to run requires a span $T_{\text{inf}} \in O(n)$.

For a while, we tried to work around this issue with various strategies for speeding up the iteration. During this process, the `cilkview` tool was invaluable for determining the true parallelism inherent in the runs. Eventually, we managed to get a reasonable amount of parallelism, $> 10\times$, but there was still a problem. That problem was that we couldn't get a measured speedup (which is always less than parallelism) better than about $1.6\times$.

As a hopeful solution to this problem, we removed the `set` reducer and wrote `bag`. At first, we were dismayed by the fact that the parallelism did not seem to have increased by much. However, although the parallelism had not increased by much, and, indeed, the measured speedup also did not increase by much (only up to about $2.3\times$ or so), the runtime of the code is now about half of what it was originally.

This makes sense, because the asymptotic running times of the various algorithms for `bag`s are much faster than their counterparts for `set`s.

4.3 Sort Nodes' Neighbors

In the search loop, when searching each node, we iterate through its neighbors in the order they are stored. At each iteration, we do a lookup into the `level` array.

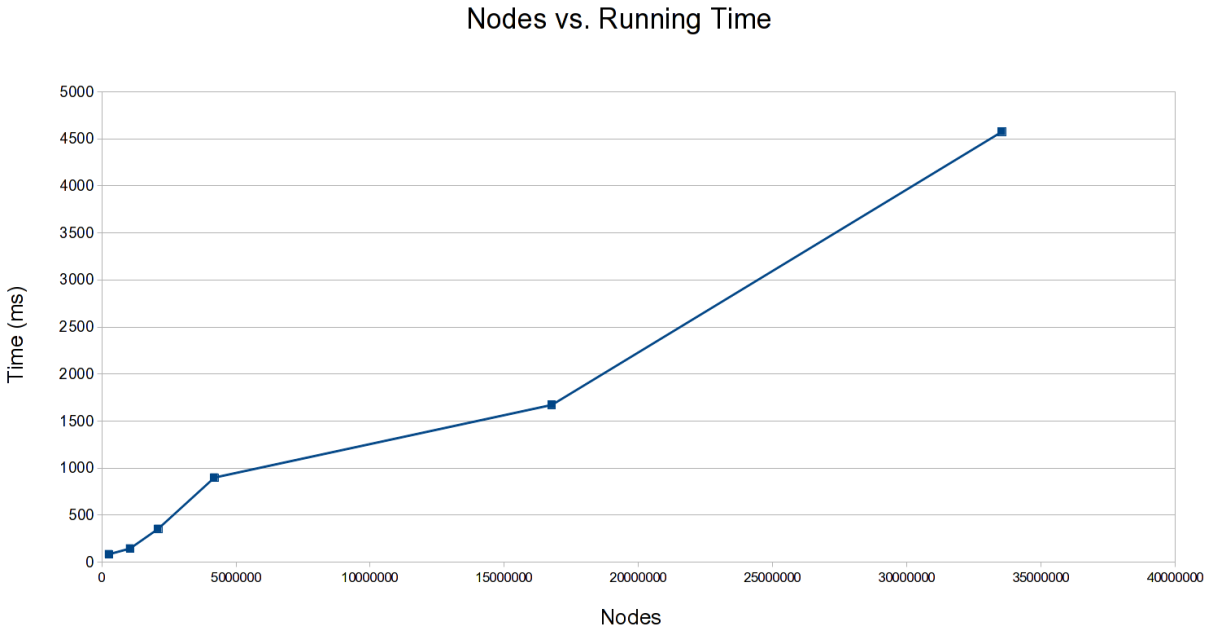
Our hypothesis was that sorting the edges belonging to each node at the very beginning would make all these accesses into `level` happen in-order, and so miss the cache less.

Upon actually testing this, we discovered that it did improve the times, but only in an extremely minor way. Specifically, the results were about 10ms slower without sorting, out of a total time of more than a second.

4.4 Final Timing Results

Once we reached the point at which we needed to stop doing experiments and writing code, we did some timing tests of our code.

Figure 1: Running time as a function of input graph size.



For $n = 2^{25} = 33554432$, we measured the TEPS rate (as defined by Graph500) to be about 27,500,000 edges/sec.

In Figure 2 (see next page), we see that speedup continues to increase up until about 8 threads.

5 Further Work

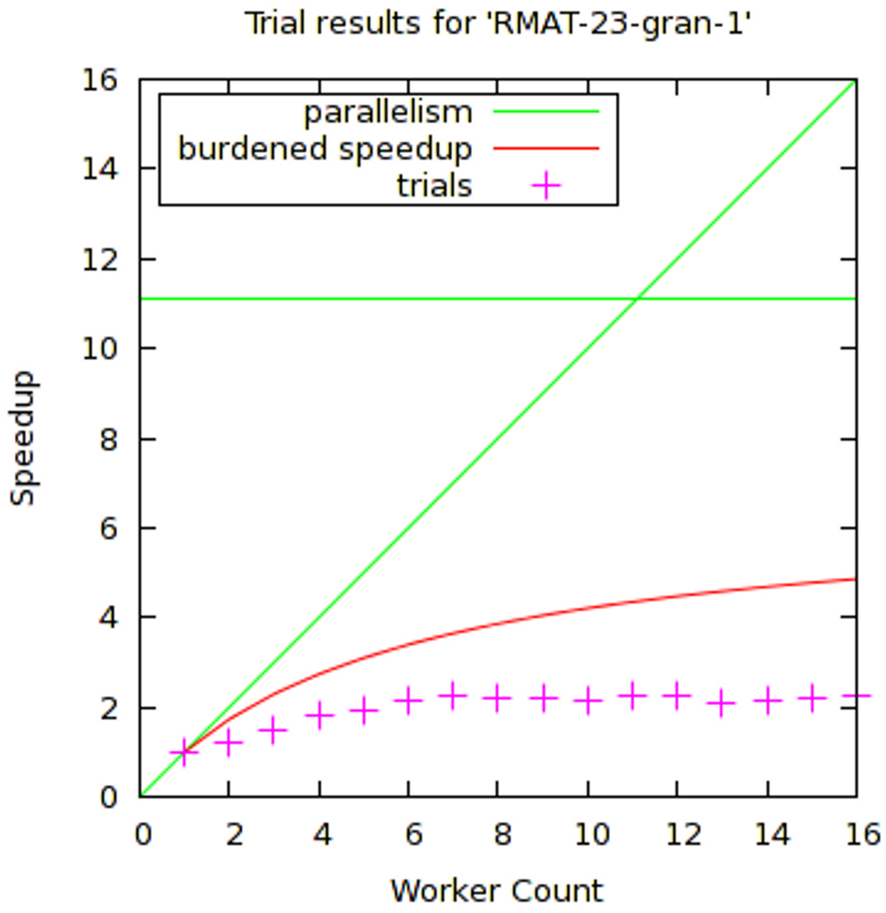
Given sufficient time, there are a still a number of open questions and improvements.

5.1 More Complicated Parallelism

One possible route to a faster search is to parallelize the breadth first search algorithm in a more creative way.

The search algorithm currently only subdivides work based on the number of nodes involved. Since the number of edges that each node has varies (these are power-law graphs), this isn't necessarily a good way to get each thread to do a decent amount of work on each call. The algorithm could take into account how much work each node will be to process and try to assign each job an equal amount of work. In order to determine how effective this

Figure 2: Expected and measured parallelism on PDAFM node.



would be, one would have to empirically observe the mean, standard deviation, and average maximum of the running times of each strand in Cilk.

The search algorithm currently performs only a single level of the search at the time. It could be changed to either do a fixed amount of work (however many levels are involved in that work being unimportant), or just operate on a fixed number of levels at a time, such as two.

These options would be much more complicated to write because of the interactions of the threads in them.

5.2 Compact Graph Representation

There are a number of ways to represent graphs with different space complexity, but the chosen representation (compressed sparse adjacency list) is the best one we know of. Fortunately, there is an obvious small improvement to make to this representation.

In particular, the Graph500 specification states that at least 48 bits must be used to

store vertex indices. However, the size of an `int` on 64-bit Linux machines is 64 bits. The obvious solution, then, is for the neighbor array, `nbr`, to contain elements of size 48 bits.

This would decrease memory usage of this array by 25%, hopefully reducing cache misses by a comparable amount. However, there would then be an overhead associated with reading an edge from the neighbor array, and some experimentation would be required to see if this is faster.

5.3 Better Bag Implementation

Obviously, the program could be made faster by replacing the $O(\log(n)^2)$ merge with a more efficient $O(\log(n))$ implementation. Of the optimizations listed here, this is by far the easiest.

5.4 Optimization for Cache Size

Like all breadth first search algorithms, ours is very sensitive to cache behavior.

All constants in the program are subject to being changed based on how they interact with the cache. As can be seen from the bitfield experiment, changing the behavior of the code can change the way it uses the cache in counter-intuitive ways.

The possibilities for changes here are far too numerous to begin to scratch the surface, but some of them follow.

- A Bloom filter could be used in place of a bitfield to accompany the `level` array. In this case, the size of the Bloom filter could be tuned to see if any speedup could be derived.

- The array containing indices of the neighbor array, `firstnbr`, could contain smaller elements than their current size of 64 bits each.

- "Sorting":

  Although sorting the elements in the next round's frontier is an $O(n \log(n))$ operation on top of the much much smaller work done by repeated splitting of `bag`s (which is only $O(\log(n)^2)$), it may be worthwhile to improve memory access patterns.

  Since the node numbers for the next round's can be summed as they are found (essentially free), their average can be considered known. Considering that they have a normal distribution (in a random graph) and their average is known, it doesn't sound too out of place to consider that there might be some kind of "almost sort" algorithm which will make them more ordered than before, although not all the way sorted at all. Obviously, the nodes being in exact order would be ideal, but any increase in their order should improve memory patterns.

  In some sense this is a time-memory tradeoff, although it is more of a tradeoff between time and memory bandwidth.

6 Conclusion

In short, parallel breadth first search is possible with decent speedup on shared memory computers, but not without taking the environment into consideration. In particular, the cost of cache misses form a significant portion of the work involved.

Our score of about 27,500,000 on the Graph500 benchmark is about 10% less edges/sec than the lowest entry in Graph500's November 2011 results. Based on our progress while working on this project, we suspect that one PDAFM node on Triton is probably capable of about 50,000,000 edges/sec (assuming that 4× speedup is possible, since there are 4 memory controllers per node).