Comparing scalable algorithms for walking all nodes of a dependency graph

COLLABORATORS			
	TITLE : Comparing scalable algorithms for walking all nodes of a dependency graph		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		June 17, 2010	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

This article describes the journey we took in order to make one area of our application scale to handle very large models, without incurring an unacceptable performance cost. It is hoped that this account may prove useful to others approaching similar problems.

1 Introduction

The latest release of our product focuses on scalability. We need to enable our users to work with much larger models, and previously they were hitting a hard limit when our 32-bit application went beyond the 2GB address space available on a normally-configured Windows machine.

Of course, there are many ways we can reduce the amount of memory we consume for each element of a model which we hold in memory, but reducing memory usage in this way does not solve the fundamental problem: if we hold it all in memory, there will always be a limit on the size of model we can work with.

In order to be able to handle models of arbitrary size, we needed to find algorithms that work with subsets of the model that are of manageable size. This article describes and compares some of the algorithms we tried in order to make our application scalable.

We began with the non-scalable approach of having everything in memory at once, and set off in the direction of scalability by implementing a naive strategy. This proved to be very slow, so we improved the execution time by making incremental changes to our approach. The algorithm we arrived at achieves bounded memory usage and acceptable performance, without imposing a heavy burden in terms of algorithmic complexity.

The models and algorithms will be illustrated in C++ using the BOOST Graph Library (BGL) [BGL]. To test our algorithms we will run them against randomly-generated models.

2 The Model

The models in our application may be thought of as directed graphs - that is, sets of "nodes" which are connected by "edges", where each edge has a direction. Cycles (paths between nodes that end up where they started) are allowed.

Figure 1 shows an example of a directed graph. The graph shown may be partitioned into three disconnected sub-graphs, indicated by the ovals. The importance of our ability to partition such graphs in this way will become clear later.

INSERT FIGURE 1 - directedgraph.svg

The edges indicate dependencies in our model. What this means in practice is that if A depends on B (i.e. the nodes A and B are connected by an edge which is directed from A to B) then it does not make sense for A to be in memory without B. Our algorithms are required to work in a scalable way within this constraint. This, of course, means that certain models (e.g. where chains of dependencies mean that all nodes are required at once) simply cannot be handled in a scalable way. Fortunately, in practice the models we work with are decomposable into smaller graphs despite this constraint [1].

[1] Note that if this were not the case we would need to revisit the requirement that it does not make sense for A to be in memory without B. In our case removing this constraint would be difficult since it would mean re-writing a large piece of functionality that is currently provided by a third-party component.

In order to test the algorithms we will discuss, we implement them using BOOST Graph Library, with the typedefs and using declarations shown in listing 1. Note that BGL prefers the term "vertex" whereas in this article we use "node" to indicate graph nodes.

Listing 1

```
#include <deque>
#include <set>
#include <boost/graph/adjacency_list.hpp>
using namespace std;
using namespace boost;
typedef adjacency_list<vecS, vecS, bidirectionalS> DependencyGraph;
typedef graph_traits<DependencyGraph>::vertices_size_type nodes_size_t;
typedef graph_traits<DependencyGraph>::vertex_descriptor Node;
```

We need many examples of possible models to evaluate our algorithms. The code in listing 2 shows how we generate random graphs. To create a graph (whose nodes are represented simply as integers) we construct a DependencyGraph object and supply the number of nodes required. To add edges we call the BGL function add_edge.

Listing 2

```
DependencyGraph create_random_graph( nodes_size_t num_nodes, size_t num_edges )
{
    DependencyGraph ret_graph( num_nodes );

    for( size_t edge_num = 0; edge_num < num_edges; ++edge_num )
    {
        add_edge( rand() % num_nodes, rand() % num_nodes, ret_graph );
    }

    return ret_graph;
}
```

At the moment, our users are working with graphs of about 700 nodes and 800 edges, so these are the numbers we will use when generating graphs for testing.

For the formal tests, we will ensure the random seed is set to a consistent value at the beginning of each test. This will mean that the various algorithms are compared against an identical set of graphs.

3 The Task

Our application performs a number of different actions. The action with which we are concerned here involves visiting all nodes and edges of the graph at least once.

In order to scale, our algorithm must limit the number of nodes that are loaded into memory at any time. In order to perform well, our algorithm must minimize the number of times each node is loaded (ideally to one) and minimize the number of loading events that happen.

For our application, the number of nodes that may be loaded at one time (before we run out of memory) is approximately 400, so that will be our limit.

The algorithm in place before we began, which we shall call "all-at-once" performs just one load event (load everything) and (therefore) loads each node only once. So this algorithm represents the best-case scenario in terms of performance [2] but fails to scale because there is no limit on the number of nodes loaded at any time.

[2] In practice, this algorithm does not necessarily perform well because it uses a lot of memory, and under certain conditions this may cause the operating system to page memory in and out from its virtual memory store, which can be extremely slow. We will not consider such practical considerations in this article: we already know that we need to limit memory usage - reducing paging would essentially mean we needed to limit it at a lower level.

To test our algorithms we need to provide a utility class that tracks their behaviour and provides them with information. The interface for this class, IGraphLoader, is shown in listing 3.

Listing 3

```
class IGraphLoader
{
public:
    virtual void LoadNodes( const set<Node>& nodes ) = 0;
    virtual unsigned int GetMaxNodes() const = 0;
};
```

IGraphLoader declares the LoadNodes method, which an algorithm may call to perform a load event. The supplied argument specifies which nodes should be loaded. In our test code, the implementation simply records the fact that the load event that has taken place so that we can provide statistics and check that all nodes have been loaded at least once when the algorithm completes.

It also declares the GetMaxNodes method, which tells an algorithm how many nodes it is allowed to load before it will run out of memory.

The all-at-once algorithm may be expressed in code as shown in listing 4.

Listing 4

```
namespace
{
    set<Node> create_all_nodes_set( nodes_size_t num_v )
    {
        set<Node> ret;
        for( nodes_size_t i = 0; i != num_v; ++i )
        {
            ret.insert( Node( i ) );
        }
        return ret;
    }
}

void algorithm_all_at_once( const DependencyGraph& graph, IGraphLoader& loader )
{
    loader.LoadNodes( create_all_nodes_set( num_vertices( graph ) ) );
}
```

The result of running this algorithm against 1000 random graphs is:

```
Percentage of failures: 100
Average number of individual node loads: 700
Average number of load events: 1
Average largest load size: 700
Average batch calculation time: 0.001329 seconds
```

The all-at-once algorithm loads each node once, and performs only one load event, but all of its runs fail because they load 700 nodes at a time, even though our maximum limit is 400. What this means is that in reality our application would crash and not be able to complete the action. This is obviously not an acceptable solution.

"Average batch calculation time" means the amount of time spent calculating which nodes to load. In this case it is very small since we do no work at all, and just load everything. This gives us a measure of the complexity of our algorithm, but is likely to be insignificant to the running time of the real program, since actually performing load events takes much longer, and that time is not included in this total.

4 The Naive Solution

In our first pass at making our application scalable, we chose a very naive solution to this problem. The solution was simply to load each node and its dependencies one by one, which we shall call "one-by-one".

The code for this algorithm is shown in listing 5.

Listing 5

```
void algorithm_one_by_one( const DependencyGraph& graph, IGraphLoader& loader )
{
    DependencyGraph::vertex_iterator vit, vend;
    for( tie( vit, vend ) = vertices( graph ); vit != vend; ++vit )
    {
        set<Node> s;
        s.insert( *vit );
        loader.LoadNodes( s );
    }
}
```

The result of running this algorithm against 1000 graphs is:

```
Percentage of failures: 0
Average number of individual node loads: 30143
Average number of load events: 700
Average largest load size: 223
Average batch calculation time: 0.019482 seconds
```

The one-by-one algorithm has a very low number of failures (in practice in our application, no failures) because it always keeps the number of nodes loaded to an absolute minimum. However, it performs a very large number of load events, and loads a very large number of nodes, since many nodes are loaded multiple times.

In our real application, this algorithm can take 30-40 minutes to complete, which is completely unacceptable for our users.

Because the process takes so long, there is almost no limit to the amount of pre-processing we should do if it improves performance. Even if our pre-processing step took a whole minute (which would imply an extremely complex algorithm) it would be a net gain if it improved performance by as little as 2.5%. However, it is desirable to keep complexity to a minimum to reduce the maintenance burden for this code.

5 Avoiding unneccesary work

The one-by-one algorithm reduces memory usage to a minimum, but has very poor performance, while the all-at-once algorithm has better performance (if it didn't crash) and far too much memory usage. We should be able to find solutions in the middle ground of the memory usage/performance trade-off.

The first observation to make is that if A depends on B, then B and all its dependents will be loaded when we load A. Thus we can improve on the one-by-one algorithm simply by skipping B - it will be covered when we do A.

In code, we have a helper function "get_minimal_set", which finds a set of nodes that cover the whole graph if you include their dependencies. It is shown in listing 6.

Listing 6

```
void remove_from_set( set<Node>& set_to_modify, const set<Node>& set_to_remove )
{
    set<Node> difference_set;
    set_difference( set_to_modify.begin(), set_to_modify.end(),
        set_to_remove.begin(), set_to_remove.end(),
        inserter( difference_set, difference_set.begin() ) );
    set_to_modify.swap( difference_set );
}

set<Node> get_minimal_set( const DependencyGraph& graph )
{
    set<Node> minimal_set;
    set<Node> covered_set;
    DependencyGraph::vertex_iterator vit, vend;
    for( tie( vit, vend ) = vertices( graph ); vit != vend; ++vit )
    {
        // Skip the node if it has been covered
        if( covered_set.find( *vit ) != covered_set.end() )
        {
            continue;
        }

        // Find all the dependencies of this node
        set<Node> this_node_deps_set;
        this_node_deps_set.insert( *vit );
        expand_to_dependencies( this_node_deps_set, graph );
```

```
        // Remove them all from the minimal set (since they are covered by this one)
        remove_from_set( minimal_set, this_node_deps_set );

        // Add this node to the minimal set
        minimal_set.insert( *vit );

        // Add its dependencies to the covered set
        covered_set.insert( this_node_deps_set.begin(), this_node_deps_set.end() );
    }

    return minimal_set;
}
```

And once we have that helper, the algorithm itself, which we shall call "skip-dependents", is simple. It is shown in listing 7.

Listing 7

```
void algorithm_skip_dependents( const DependencyGraph& graph, IGraphLoader& loader )
{
    set<Node> minimal_set = get_minimal_set( graph );

    // Load each node in the minimal set one by one (similar to
    // "one-by-one" algorithm).
    for( set<Node>::const_iterator msit = minimal_set.begin();
         msit != minimal_set.end(); ++msit )
    {
        set<Node> s;
        s.insert( *msit );
        loader.LoadNodes( s );
    }
}
```

The result of running this algorithm against 1000 random graphs is:

```
Percentage of failures: 0
Average number of individual node loads: 9785
Average number of load events: 223.188
Average largest load size: 223
Average batch calculation time: 0.038484 seconds
```

The skip-dependents algorithm is functionally identical to one-by-one - it simply avoids doing unnecessary work. It works much faster than one-by-one, and fails just as infrequently.

However, the performance of skip-dependents is still poor. We can do much better by handling nodes in batches.

6 Batches of nodes

We need to reduce the number of nodes that are loaded more than once, and reduce the overall number of load events, so we gather our nodes into batches, and process several at once. Each batch with all its dependencies should be smaller than the maximum number of nodes we can fit in memory, but (if we ignore the performance impact of using a large amount of memory) we should load as many nodes as possible within this constraint. A very simple application of this idea is an algorithm we will call "arbitrary-batches".

7 Arbitrary batches

The arbitrary-batches algorithm handles the nodes in an arbitrary order. It starts with the first node and calculates its dependency set. If the set is smaller than the maximum number of nodes, it adds the second node and its dependencies to the set and again

compares its size with the maximum. When the number of nodes goes over the maximum, it stops and reverts to the previous set, and loads it. It now continues with the node that was rejected from the previous set and repeats the process until all nodes have been loaded.

To implement this algorithm we will need a small helper function "get_dependency_set_size", shown in listing 8.

Listing 8

```
size_t get_dependency_set_size( const set<Node>& set_to_examine,
    const DependencyGraph& graph )
{
    set<Node> set_with_deps = set_to_examine;
    expand_to_dependencies( set_with_deps, graph );
    return set_with_deps.size();
}
```

The main code for arbitrary-batches is shown in listing 9.

Listing 9

```
void algorithm_arbitrary_batches( const DependencyGraph& graph, IGraphLoader& loader )
{
    set<Node> minimal_set = get_minimal_set( graph );

    set<Node> current_set;
    set<Node> proposed_set;

    // current_set is the set that currently fits into our maximum
    // size (or a set of size 1, which may not fit, but is as small
    // as we can go).

    // Loop through all the elements.  For each element, try to add
    // it to the proposed_set.  If it doesn't fit, load current_set.
    for( set<Node>::const_iterator msit = minimal_set.begin();
         msit != minimal_set.end(); ++msit )
    {
        proposed_set.insert( *msit );

        // If proposed_set contains more than one element, and
        // is too big, reject it and load current_set.
        if( proposed_set.size() > 1 &&
            get_dependency_set_size( proposed_set, graph ) >
                loader.GetMaxNodes() )
        {
            loader.LoadNodes( current_set );

            // Now reset to form the next batch
            proposed_set.clear();
            proposed_set.insert( *msit );
            current_set.clear();
        }

        // proposed_set is now acceptable, so make current_set
        // identical to it by adding the node.
        current_set.insert( *msit );
    }
    if( !current_set.empty() )
    {
        loader.LoadNodes( current_set );
    }
}
```

The result of running this algorithm against 1000 random graphs is:

```
Percentage of failures: 0
Average number of individual node loads: 1500
Average number of load events: 4.065
Average largest load size: 399
Average batch calculation time: 0.051833 seconds
```

This shows a huge improvement over the skip-dependents algorithm. We tend to use almost as much memory as we are allowed, but perform far fewer load events, and load far fewer nodes in total, so the execution time is much smaller.

8 Optimal batches?

Of course, we can do better than batching nodes in an arbitrary way. We can achieve the absolute optimum batching arrangement by examining all possible batches of nodes and picking the arrangement that results in the smallest number of batches that are within the maximum size. We will call this algorithm "examine-all-batches".

An algorithm for finding all partitions of a set (another way of saying all ways of batching our nodes) may be found in [Knuth] ("Algorithm H"), along with a discussion of how many partitions we will find for different sizes of set. Actually implementing this algorithm, however, could be argued to be a waste of time, since the number of possible batching arrangements of 700 nodes is 4767951360475272425825869131310359104962555270612535401321050276195337861059635830616333417663828373896486116 [4], which is a lot.

[4] The number of ways of partitioning different size sets are called Bell numbers. There is lots of interesting discussion of them in [Knuth]. The numbers shown here were calculated using a small Ruby program found on Wikipedia at http://en.wikipedia.org/-wiki/Bell_number.

Of course, the number above is unfair, since we already have a way of reducing the effective number of nodes by skipping dependencies. When we run the skip-dependents algorithm, we find that the average size set of nodes needed to cover the whole graph is 223. The number of possible batching arrangements of 223 nodes is 10040496258227859512495184257199289182868965124946595195 which is a lot.

So, we need to find a compromise that picks good batches without waiting around to find the absolute best one.

9 Best partner

One compromise is to work through in an arbitrary order, but for each node we encounter, pick the best possible partner for it from the other nodes. This means in the worst case we need to evaluate (N^2)/2 unions. We will call this algorithm "pick-best-partner".

The code for pick-best-partner is shown in listing 10.

Listing 10

```
void algorithm_pick_best_partner( const DependencyGraph& graph, IGraphLoader& loader )
{
    // This set contains nodes that have not yet been processed
    set<Node> remaining_set = get_minimal_set( graph );

    while( !remaining_set.empty() )
    {
        // This set is what we will load
        set<Node> current_set;

        for( set<Node>::iterator it = remaining_set.begin(); it != remaining_set.end(); )
        {
            current_set.insert( *it );
            remaining_set.erase( it );

            it = remaining_set.end();
            size_t best = loader.GetMaxNodes(); // Even our best match must be below the ↩
                limit
```

```
            for( set<Node>::iterator i = remaining_set.begin(); i != remaining_set.end();  ↩
                ++i )
            {
                set<Node> current_plus_one( current_set );
                current_plus_one.insert( *i );

                size_t dep_size = get_dependency_set_size( current_plus_one, graph );

                if ( dep_size < best )
                {
                    best = dep_size;
                    it = i;
                }
            }
        }

        // Load the best set we have found.
        loader.LoadNodes( current_set );
    }
}
```

The result of running this algorithm against 1000 random graphs is:

```
Percentage of failures: 0
Average number of individual node loads: 1104
Average number of load events: 3.042
Average largest load size: 398
Average batch calculation time: 2.73926 seconds
```

This algorithm produces a good result, with fewer load events being needed than for the arbitrary-batches algorithm. However, it is considerably more complex than the other algorithms, which makes it harder to understand, and it is also more computationally expensive. It is worth considering simpler alternatives.

10 Order by size

The last algorithm we considered is called "order-by-size". It is computationally simple, and produces results comparable with pick-best-partner.

The order-by-size algorithm is designed to exploit a common property of graphs, which is that they often consist of relatively separate clusters of connected nodes. As illustrated in figure 2, let us imagine that we have a cluster of nodes that is depended on by several others: A, B and C. The first thing to note is that there is no need for us explicitly to load any of the nodes in the cluster, since they will automatically be loaded when any of A, B or C is loaded. In fact, using the skip-dependents algorithm, they will all be loaded three times - once for each of A, B and C.

INSERT FIGURE 2 - cluster.svg

It is obviously a good solution in this case for us to batch A, B and C together, and the pick-best-partner algorithm is quite likely to do this.

If we assume that our graphs are of a structure like this, however, there is a simpler algorithm that may well result in A, B and C being batched together. It comes from the observation that the set of dependencies of A, B and C are of similar size. Thus if we order the nodes by the size of their dependency set, and then perform the same algorithm as arbitrary-batches, we are likely to place A, B and C near each other in our ordering, and therefore to batch them together. This algorithm is much less complex and computationally expensive than pick-best-partner so if it works well, it may be more helpful when we have very large graphs.

The code for order-by-size is shown in listing 11.

Listing 11

```
typedef pair<size_t, Node> size_node_pair;

struct NodeToPair
{
    NodeToPair( const DependencyGraph& graph )
    : graph_( graph )
    {
    }

    /** Given a node, returns a pair of the size of
     *  its dependency set and the node. */
    size_node_pair operator()( const Node& v )
    {
        set<Node> tmp_set;
        tmp_set.insert( v );
        return size_node_pair(
            get_dependency_set_size( tmp_set, graph_ ), v );
    }

    const DependencyGraph& graph_;
};

void algorithm_order_by_size( const DependencyGraph& graph, IGraphLoader& loader )
{
    set<Node> minimal_set = get_minimal_set( graph );

    // Create a set of nodes sorted by their dependency set size.
    NodeToPair v2p( graph ); // Create a converter v->(size,v)
    set<size_node_pair> sorted_set;
    transform( minimal_set.begin(), minimal_set.end(),
        inserter( sorted_set, sorted_set.begin() ),
        v2p );

    // The rest is identical to arbitrary-batches, except looping
    // through sorted_set instead of minimal_set.
    set<Node> current_set;
    set<Node> proposed_set;
    for( set<size_node_pair>::const_iterator ssit = sorted_set.begin();
         ssit != sorted_set.end(); ++ssit )
    {
        proposed_set.insert( ssit->second );
        if( proposed_set.size() > 1 &&
            get_dependency_set_size( proposed_set, graph ) >
                loader.GetMaxNodes() )
        {
            loader.LoadNodes( current_set );
            proposed_set.clear();
            proposed_set.insert( ssit->second );
            current_set.clear();
        }
        current_set.insert( ssit->second );
    }
    if( !current_set.empty() )
    {
        loader.LoadNodes( current_set );
    }
}
```

The result of running this algorithm against 1000 random graphs is:

```
Percentage of failures: 0
```

```
Average number of individual node loads: 1104
Average number of load events: 3.008
Average largest load size: 397
Average batch calculation time: 0.056774 seconds
```

This algorithm performs almost exactly as well as pick-best-partner on the random graphs against which it was tested, and is easier to understand and faster to run.

11 Conclusions

After our investigation, the order-by-size algorithm was chosen by our team to choose the batches of nodes to be processed in a scalable way. The next release of our product is out soon, and scales much better than the previous one.

I hope that this glimpse into the thinking process we went through as we worked to make our product scale to handle larger models provides an insight into the different ways of thinking about algorithms that are needed to address a problem in a scalable way.

I feel that our results show that sometimes the most mathematically complete algorithm can be overkill, and a much simpler approach can be a better solution. We chose a solution that satisfied our requirements, executed quickly, and avoided introducing undue complexity into the code base.

12 References

[1] [BGL] http://www.boost.org/doc/libs/release/libs/graph/

[2] [Knuth] Knuth, Donald E. "Combinatorial Algorithms", in preparation. Downloadable from http://www-cs-faculty.stanford.edu/~knuth/taocp.html