overload 97

JUNE 2010 £3

Scalable Graph Coverage

Optimising data processing is often about compromise. We try to find the right balance.

Socially Responsible Recruitment

Finding talented people is hard. Here's an underappreciated source.

The Functional Student

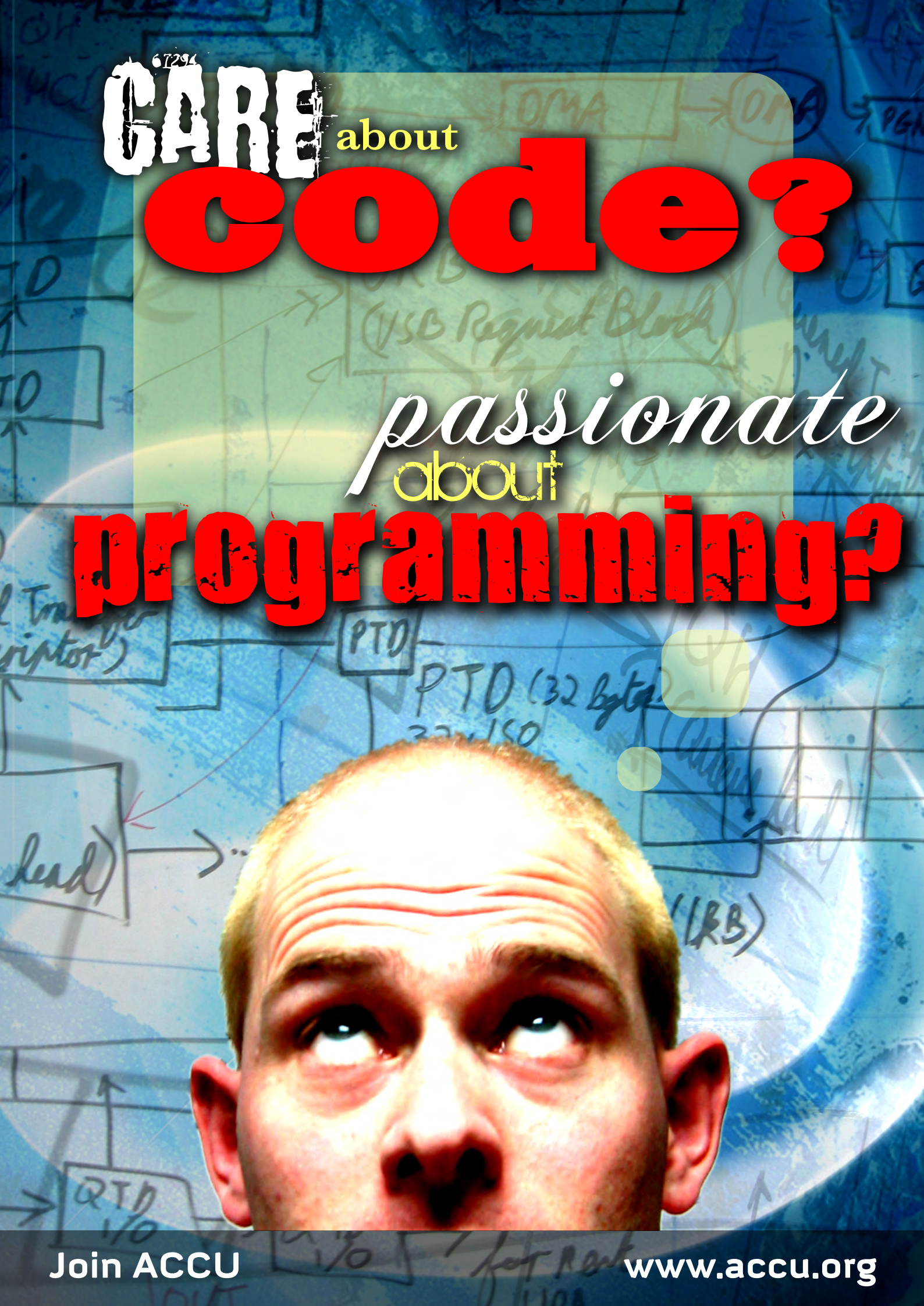
We model a game of six integers in F#. And in C++.

Single-Threading: Back To The Future?

The 'multi-core revolution' is finally here, and software development is not getting any easier. We look at problems with multi-threading, and some strategies for mitigating it.

A magazine of ACCU

ISSN 1354-3172


CARE about code?
passionate about programming?
Join ACCU
www.accu.org

OVERLOAD 97

June 2010

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 98 should be submitted by 1st July 2010 and for Overload 99 by 1st September 2010.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of ACCU For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

The Art of the Possible

In polite company you should never talk about religion or politics. Ric Parkin makes an exception.

I've started to write this on the evening of The Queen's Speech at the opening of the new parliament, where she announces the priorities and the legislation that the government will bring forward over the next eighteen months (traditionally this happens after the summer recess, so is a bit longer than the usual year, due to the change of government). Normally you shouldn't talk politics as it tends to generate more heat than light, and it's one of the few off-topic subjects on accu-general, that otherwise free 'pub conversation', but this has been a very interesting campaign from a technology point of view, and the consequences of the change of government will affect certain IT sectors quite heavily, so I think I can safely make some comments without stepping over the line into controversy.

The first Internet election?

One thing that seemed very different this year compared to the last general election was the role the Internet, and in particular the new social networking sites and services. This shouldn't have been too much of a surprise after the US presidential elections of 18 months ago which also saw a big change. Instead of faintly laughable party websites of five years ago, there were much slicker campaigns, with much more 'community' blogs and sites, Facebook to find interested groups and show symbolic support, Twitter to quickly disseminate news, opinion, and rebuttal.

So what sort of things did, and didn't, make much of an impact? Well, I think it depended an awful lot on who you were and what sort of media you consumed! For myself, I tended to watch some TV news, read some online news sites of both television stations and newspapers, and noticed what Facebook friends were saying. I'm not on Twitter, but some friends have theirs hooked up to Facebook, so I got some of that. I don't tend to read political blogs either, although this time I do admit to obsessively reading about the vast number of polls that were being taken, such as the UK Polling Report [UK], and the attempt to predict the final outcome by people like Nate Silver, who had so successfully predicted the result of the US elections [538].

But oddly enough, the main places I found out about what was happening on FB, Twitter, or many of the more partisan blogs, was from the 'traditional' media, who seemed to fill the chasms of 24-hour rolling news with up-to-the-minute posts of not just what the official campaigns were doing, but also what the online army of bloggers, tweeters and commentators, whether they were partisan insiders or interested observers, opinionated or objective, professional or amateur, the man on the Clapham omnibus [clapham], or sometimes the downright unnerving one-issue fanatics.

Sometimes these sources of information broke major events, but overall I found them to be remarkably poor despite the mainstream media's obsession with them. Sometimes the event would be a major story one minute, only to be forgotten about a few hours later. And I wondered why this should be so.

So here's my wild ideas on why this was the first UK Internet election, and why this was not necessarily a good thing.

The first is, there's now so much pressure to get any story out there or someone else will scoop you, and there's so much air time or web space to be filled (and yet it's so cheap to do), that taking your time and getting the quality control right has pretty much gone. In the past, a news story would have to get past a stern editor who'd make sure it was at least plausible, vaguely accurate, and ideally being reasonably non-partisan. That works fine for a main news bulletin or a daily or Sunday newspaper, but the Internet and 24 hour news means you can get the story out fast without pausing for thought. In particular the more 'opinion' media, such as blogs, Facebook statuses and Twitter – you just type and press a button, and it's out there. If you regret it five minutes later, then it's too late. If it turns out to be wrong but it's already been copied and forwarded and commented on by others, then it's too late. If it's merely hearsay (or you want to start a rumour) then there's often no one else to ask 'are you sure?'. It's very hard to bolt that stable door, and it's even harder to get the horse back. It can be especially hard to get people to abandon a story when they have an emotional attachment to a position that it reinforces, even when the story can be shown to be false. This is why the main parties now have 'rapid rebuttal units' to spot when something damaging is reported, and get out a neutralising counter-story before it gets very far (or at least onto that night's main news).

I think this led a lot of the 'new' media reporting to be remarkably trivial, or stories that only last a short amount of time: the time delay and bandwidth limitations used to allow quality control to hone or outright reject the second-rate stories, but now they're published immediately, and the fixes come afterwards.

Another insidious effect of the social networks was a ratcheting up of 'group think', where your ideas and opinions are confirmed and strengthened by them being agreed with by the people around you. The trouble is that you tend to surround yourself with similar people with similar ideas, so you get into a circle that reinforces your ideas and preconceptions. It is possible to get outside viewpoints, but you have to try harder and it is never comforting to challenge yourself. This sort of thing is not limited to politics – the Internet has allowed a wide range of social interactions with such a huge scope to meet people with similar opinions you'd otherwise have never have come across. This is most noticeable in the more 'fringe' groups, where all sorts of strange ideas can be shared without having to check them, and where any counter-evidence merely confirms how hard the conspiracy is trying to suppress The Truth. But while these are obvious examples of Group-Think (at least to those outside) we should always be aware that we too are going to be prone to such distorting self-selection. This is why self-correction mechanisms

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

such as peer-review are so important. By setting up a system where proving an idea wrong leads to success we create incentives to detect and discard faulty ideas, and those that remain tend to be the best supported (so far).

But it isn't all bad. Despite their flaws, social media have allowed many more people to feel like they were involved (one of the criticisms of our First-Past-The-Post electoral system is that most elections are decided by a relatively small number of floating voters in marginal constituencies – if you're not one of them then you are pretty much ignored as the parties concentrate their efforts on those key people). Also the speed that news can be broken and spread can be a terrifically powerful force. Perhaps my fears are just that we're still learning about how these media are best used, after all it was only a few months ago that Twitter was just a pointless way of getting mundane information about friends and celebrities, until it became a vital way of spreading information during social unrest where it could bypass and outmanoeuvre state controlled systems. We haven't needed to use it for that, and yet it still is a remarkably fast way of communication. A possible explanation came to light recently, as it was suggested that Twitter in particular has many properties of a Small World network, like many social groupings, yet is more tightly interconnected, so news can travel very quickly [Sysomos].

It wasn't just the new online world that was shaken up during this election: for the first time we had TV debates involving the party leaders. This really shook things up for a while, although perhaps ultimately not as much as people thought at the start. Again, these are early days and we're learning fast, although some of the attempts at 'instant feedback' were delightfully rubbish, my personal favourite was the company trying to gauge sentiment by analysing traffic on Twitter, which found high levels of dissatisfaction with the Liberal Democrats because the computer couldn't understand the joke meme of blaming Nick Clegg for all sorts of bizarre things [NickClegg].

And now what?

So what are the new government going to do? There's quite a lot of IT related promises, so it is worth having a think about them. The most high-profile announcement and one of the ones to have a quick impact, is the announcement to cancel ID cards, dismantle the National ID Register behind it, and reform or scrap other 'big databases' such as Contact Point and the DNA database. That's going to have a profound change of emphasis on what should be done with data generated by you, which will mean that there will be far fewer large IT contracts for building and running such things. This is already having an effect, as can be seen by a quick look at the share prices around the election of outsourcing companies such as Capita.

Two other effects – there looks to be a change in Capital Gains Tax to be announced soon. Details are still being decided as I write, but there could be great implications for people in small start-up companies who use share options as a major incentive, in particular if the rate is increased and taper relief changed. The other one is a bit more subtle – there was mention about making a level playing field for open-source software [Programme]. It is not clear yet what this actually means, but it seemed odd as earlier both Labour and the Conservatives were also talking about open source standards [Register]. I think the difference is important – open standards mean that anyone could write a system to use some data, and you would not be restricted to the original supplier. I hope this was just an oversight during the frantic negotiations that were happening at the time, but time will tell.

A couple of anniversaries

And finally, a couple of notable anniversaries. It is a decade ago that the first 'rough draft' of the human genome was publish to great acclaim [Observer]. A great achievement that was particularly reliant on computing power – the sheer number of bases involved are staggering, and many of the techniques used involved breaking the genome in short pieces, sequencing those pieces, then reassembling them (a so-called Shotgun approach). Working out where to place each of those pieces was a massive computing problem in its own right – think of it as a jigsaw puzzle with several billion pieces. Since then computing power, the storage required, and the tricks used to solve problems specific to genetics, have progressed massively. Similar to other great data-producing experiments such as the Large Hadron Collider, much of modern science relies on massive computing power to process swathes of data, or model fantastically complex systems to gain insights beyond our ancestors dreams.

The other anniversary is more mundane – I've been Overload editor for two years now. It's been great fun, but I have to give kudos to the many people who do the real work – I just coordinate and write the editorial. But is always good to get feedback, and I'm pleased to say the last couple of issues have really got things going. Richard Harris' article on the Countdown Numbers Game seems to have struck a chord, resulting in two readers to offer very different solutions which makes for some fascinating compare and contrast. And my last editorial on recruitment also inspired an article in this issue on Socially Responsible Recruitment, and a couple of long threads on accu-general on the changed role of job-centres (I'm pleased to report they have improved tremendously since I last used one), and how to set quizzes to get the right sort of applicant which really showed that there is not one true answer, and that it all depends on your own situation. As a consultant would say: 'It depends'.

References

[538] http://www.fivethirtyeight.com/

[clapham] http://en.wikipedia.org/wiki/The_man_on_the_Clapham_omnibus

[NickClegg] search for #nickcleggsfault or #iblamenickclegg

[Observer] http://www.guardian.co.uk/theobserver/2010/may/30/dna-human-genome-project-sanger

[Programme] http://programmeforgovernment.hmg.gov.uk/government-transparency/

[Register] http://www.channelregister.co.uk/2010/03/11/tory_tech_manifesto/

[Sysomos] http://sysomos.com/insidetwitter/sixdegrees/

[UK] http://ukpollingreport.co.uk/blog/

Scalable Graph Coverage

Optimising data processing is often about compromise. Andy Balaam finds the right balance.

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.

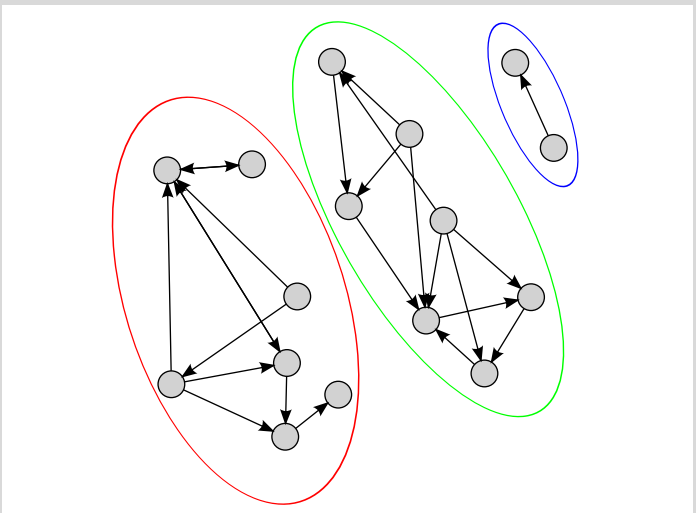


Figure 1

Andy Balaam is happy so long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his many open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net.

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.

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].

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.

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`.

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.

```
#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;
```

Listing 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 perform well, our algorithm must minimize the number of times each node is loaded

```
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;
}
```

Listing 2

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.

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.

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.

`IGraphLoader` declares the `LoadNodes` method, which an algorithm may call to perform a load event. The supplied argument specifies which

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

Listing 3

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.

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

```
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 ) ) );
}
```

Listing 4

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.

In our first pass at making our application scalable, we chose a very naive solution to this problem

```
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 );
  }
}
```

Listing 5

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.

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.

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.

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.

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

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.

```
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;
```

Listing 6

Each batch with all its dependencies should be smaller than the maximum number of nodes we can fit in memory

```
  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;
}
```

Listing 6 (cont'd)

```
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 );
  }
}
```

Listing 7

```
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();
}
```

Listing 8

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

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'.

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.

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

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.

In theory we could find the optimum batching arrangement

```
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 );
  }

}
```

Listing 9

Optimal batches?

Of course, we can do better than batching nodes in an arbitrary way. In theory we could find the 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 476795136047527242582586913131035910496255527061253540132105027619533786105963583061633341766382837389648611612771677592830522886078735705416705452453008326983118350278856293246415442825525872132767880703611269517916410699389083898617416296436067998980619863172119737049355905570755067439951890427106073603254568375246887867651403802347197578978486150503432889024029158705125174242857617441606128695723264050292389444854049797955371493350209036229336528542064073126866607390273651430141623552713645986307716444010184419181435356236135376448679116971311289633490791588028597283628657999477361757107291731808263591659810246723306278903986141179776725930061451455179970017669748814194612040515465108922158960995325054798370278578711015636790696712476107846283213465931511632378594732942053291554519641861084701081135043648742081247169650980824359937460375243860549910565278335454952329848083353096239237389959263098234831832139223896341251478082772611627030798356337930215506087194224901166091100188464678604522172032294810342207269225002674386180154345829514234908836444020438335175453265053182443932474694815803283799465734979460310454816751416300950275207208963066740799248661317055264382949360712571626661026520510975817417654808336584166409476785225038811885, which is a lot.[3]

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 100404962582278595124951842571992891828689651249465951918788712771148462387607643604154059186569653443630271583320290432899249190157369402894454554210598884406055771977737151420536445318453422753766960034699363738064393880166051939418461673101556556907490355273. 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.

```
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 );
  }
}
```

Listing 10

618913370747221579255542494272489057981277287118965 80, which is a lot.

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

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.

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.

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.

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.

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
....
```

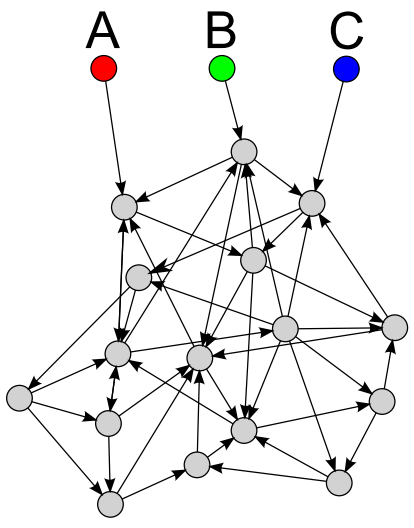


Figure 2

```
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 );
  }
}
```

Listing 11

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.

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. ■

Acknowledgements

Thanks to Charles Bailey, Edmund Stephen-Smith, Ric Parkin and the reviewers for comments, correction and code.

References

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

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

EXCEPTIONAL REWARDS, CENTRAL OXFORD

Why does a leading Quant Fund want to recruit leading software engineers?

You're developing software that's highly efficient and never fails. We're using software to generate positive returns in chaotic financial markets. We think we should talk.

Who Are We?

OxFORD ASSET MANAGEMENT is an investment management company situated in the centre of Oxford. Founded in 1996, we're proud of having generated positive returns for our investors each year, including 2008, particularly as many assets are managed for pensions, charities and endowments. We blend the intellectual rigour of a leading research group with advanced technical implementation. We like to maintain a low profile, nobody comes to work in a suit and we are a sociable company.

What Do We Do?

We use quantitative computer-based models to predict price changes in liquid financial instruments. Our models are based on analyzing as much data as we can gather and we actively trade in markets around the world. As these markets become more efficient, partly because of organizations like ours, we need to develop improved models in order to remain competitive. Working to understand and profit from these markets provides many interesting mathematical, computational and technical challenges, especially as markets become increasingly electronic and automated. We enjoy tackling difficult problems, and strive to find better solutions.

Who Do We Want?

Although most of us have advanced degrees in mathematics, computer science, physics or econometrics from the world's leading universities and departments, we are just as interested in raw talent and will consider all outstanding graduate applicants. We expect all prospective candidates to work efficiently both in a team environment and individually. We value mental flexibility, innovative thinking and the ability to work in a collaborative atmosphere. No prior experience in the financial industry is necessary. We want to hear from you if you are ambitious and would relish the challenge, opportunity and excellent compensation offered.

Software Engineers

We are seeking outstanding software engineers to develop and maintain system critical software. You will be responsible for all aspects of software development on a diverse range of projects, such as automating trading strategies, integrating third party data into our system and the development of data analysis tools.

You will have the following:

- A high quality degree in computer science or related discipline
- Several years C++ experience, including the use of the STL and Boost
- The ability to write high performance code without sacrificing correctness, stability or maintainability
- A good understanding of Linux, scripting, working with large numerical data sets and large scale systems
- Mental flexibility, innovative thinking and the ability to work quickly in a collaborative atmosphere

Any experience in the following areas would be advantageous: numerical analysis, optimisation, signal processing, statistics, machine learning or natural language processing.

Benefits

Health Insurance (including family)
Pension Scheme
Life Insurance

Closing Date
Ongoing

Start Date
Spring 2010

How to apply

Please email or post CV with covering letter, stating which position you are applying for to:

Dr Steven Kurlander
Oxford Asset Management
Broad Street
Oxford
OX1 3BP
United Kingdom

accu@applytooxam.com

+44 1865 258 138

www.oxam.com

OXFORD ASSET MANAGEMENT

Socially Responsible Recruitment

Finding talented people is hard. Ian Bruntlett highlights an under-appreciated source.

Some companies portray themselves as having a social conscience – usually by explaining how 'green' is the power that the company uses. There is another angle to being a socially beneficial company, that of supporting disabled people – part of society's forgotten population. That category is split into two more categories: physical disability and mental disability. I will concentrate on the psychiatric side of things, outlining a typical process of treatment, and then explore a few of the ways socially aware companies can give their support.

Initially this article focussed on the use of people with psychiatric problems in software houses. Upon consideration this article could be applied to people within the I.T. industry and beyond.

People start off being patients, staying on a psychiatric ward some time, with some activities run by the staff. After being patients, people are called 'clients' or 'service users' when they get moved out of the hospital and into the community.

Most of my information about mental health is based on first hand experience and by being acquainted with other patients. There are NHS sponsored 'gatherings' where NHS staff meet patients to learn from their experiences. See the references section at the end of this article for additional sources of information.

Official psychiatric articles usually have a high level of statistics and are kept hidden behind paywalls on the internet. For the purposes of this article I have drawn on personal experience and shown my key worker draft copies of this article.

Physical disability is something I haven't personally experienced. However, most buildings may need a ramp for wheelchair access and the provision of stair lifts for the physically disabled. If you're missing an arm or a leg then most people in that situation get by OK. Contact (see References at the end) provides stair lifts for people to use. It is a 'visible' illness and people find it easier to relate to.

Mental disability is something I have and continue to have experience of. There is an escalating scale of illness ranging from anxiety or depression all the way up to manic depression (now known as bipolar disorder) and schizophrenia. These tend to be 'invisible' illnesses and some people find it harder to relate to.

If mental ill-health bestowed only disadvantages to the sufferer and their social group, diseases like bipolar disorder and schizophrenia would have disappeared by now. It has been said that schizophrenics are divided into two separate camps – either those that can barely tie their shoe laces together and those that are brilliant thinkers. I know of one person – RC – who has bipolar disorder yet he held down a high-flying career in commerce and is very good with poetry. I have been given anecdotal evidence that a family with autism had a flair for maths [Times] at a University level of education. I have also read that Richard M. Stallman, founder of the Free Software Foundation suffers from Asperger's syndrome (which in turn is linked to autism).

Ian Bruntlett Ian is a volunteer system administrator for a mental health charity called Contact (www.contactmorpeth.org.uk). As part of his work, Ian has compiled a free Software Toolkit (http://contactmorpeth/wikispaces.com/SoftwareToolkit).

Schizophrenia

My name's Ian [Bruntlett] and I have schizophrenia. I experience negative symptoms and positive symptoms.[NHS] [Wikipedia] Positive symptoms include hallucinations (auditory mainly, sometimes visual), thought disorder, delusions and cognitive impairment. Negative symptoms include withdrawal from social activities, emotional flatness, social apathy.

After nearly a decade of treatment courtesy of the NHS – and with a fair amount of therapy – you usually build up some idea of what is real and what is suspect. A foundation of getting to grips with schizophrenia is the NHS care in the community programme, implemented by a Community Assertive Outreach Team (CAOT), with social activities (e.g. walking group) to prevent the kind of social exclusion that people experience.

Psychosis vs reality

When I was first admitted onto a hospital ward, I was psychotic – I had lost track of reality. The lack of stimuli was a positive aspect of the ward while I struggled with hearing voices and delusions. It took three months to get me stable and onto medication. After that I was allocated a CPN (Community Psychiatric Nurse) and a Social Worker and discharged back into my own flat in the community.

A variety of approaches were taken to draw me gently back to reality. I agreed to speak to student psychiatrists about my experiences. I had regular conversations with staff (a senior nurse, key workers). The medication helped but the key factor was the NHS staff.

The NHS have a variety of strategies that are used to help patients. One key strategy is 'distraction' from voices, visions and disturbing thoughts. My main key worker on rehab (then East Loan), 'CC', helped me by talking about my strange beliefs, voices in my head and paranoia by introducing me to a variety of 'distraction' categories. We created an emergency credit card for use when going out into the community. It stated that I had a mental health problem, and provided emergency phone numbers. On the other side it listed the following 'coping strategies':

- Ground self in present (consciously keeping a tight hold of reality).
- Think of consequences of actions.
- Try reading a book.
- Watch T.V.
- If on a bus, look out of a window, move seats.
- Phone staff.
- Breathing exercises (to regain a sense of calm).
- Remember all achievements, positive things that have happened.
- Use PRN medication – supplementary medication taken when required.

we take in unwanted PCs, fix & refurbish them and give them away to people with mental health problems, their carers or their children

Cognitive impairment

Cognitive impairment is such an innocent phrase. Sometimes you lose track of a conversation. Sometimes it takes seemingly forever to understand something as simple as a water bill. Sometimes things that took a day (reading a paperback novel) now takes a lot longer (3 months, at its worst, now down to 3 or 4 days). Coping with an illness like schizophrenia is a full time occupation and this can lead to existing skills getting a bit rusty and missing out on more modern developments. So how does cognitive impairment start?

One cause of cognitive impairment is staying on an NHS psychiatric ward itself, for say... eight months solid. They provide a calm, stress free and low stimulus environment. Most people just sit around drinking NHS tea, chatting, watching TV or listening to the radio – instead of working on projects, communicating via emails and reading books.

There is no such thing as 'keeping busy' on a ward. With the help of a T.I. (Technical Instructor), I was allowed to access the internet using the Occupational Therapy department's patient access computer. I was able to keep track of news by reading The Register, to keep in touch with friends using a Hotmail account, and reading the messages on the accu-general mailing list. I did try to run a table top RPG on the ward – *Call of Cthulhu* – but the would-be players were moved to another unit because they were caught smoking cannabis.

Another cause of cognitive impairment is medication. Medications usually have some very nasty side effects. Mine – clozaril – can attack white blood cells which means I have regular blood tests to check that this is not happening. Typical medication can also lead to weight gain (eventually resulting in diabetes), lethargy and poor concentration. Supplementary medication can also have side effects. I took some – diazepam – and it made me so tired I just ***had*** to sit down and wait for the effects to dissipate.

A final cause is moving onto a specialist rehabilitation unit. Rehab is not just for drug users and has a variety of workers – Project Workers, Technical Instructors, Occupational Therapists, Mental Health Nurses, Psychologists and Psychiatrists. With their help the patients are prepared for life outside of the hospital. This is where things start getting better. Patients live on the hospital grounds in a small clustered community of single person flats. Patients have their week structured in the form of a weekly planner – various activities are arranged (food shopping, travelling to and from voluntary work). There is a strategy called *graded exposure* which is applied to many things but one instance in particular relates to how patients move from living on a hospital ward to living in the community. First you go home for a few hours with a member of staff once a week, then you go there alone for a few hours, then you go for a day or two and eventually you are moved out of rehab and into the community.

Productive activity

Once a patient has been stabilised and the right medications worked out, attention focuses on either going onto a longer stay unit followed by being moved into the community or going straight out into the community.

For a long time (at least a century), the NHS and the charities that preceded it have encouraged its stable psychiatric patients to engage in some activity – usually voluntary work. There are several reasons why the NHS encourages its patients to engage in these kinds of activities because an evidence based approach has shown it to be therapeutic and it has been observed that sitting around all day is bad for physical health – some form of activity is helpful. There used to be a farm on the hospital's grounds but that has long since gone.[StGeorge's] These days patients work at garden centres, in the Kiff Kaff (the hospital's café) or (in my case) in Contact's computer project. Salary is a matter of a few pounds a day or, in my case, a bacon sandwich a day. And patients benefit by experiencing the discipline of working and, perhaps, gaining skills like cooking or gardening. After a period of time, the patients also experience the satisfaction of doing something useful with their lives, having something tangible to account for their time.

The only cognitive exercise I experienced in the hospital was the weekly ward quiz. The Occupational Therapy department works towards preparing the patient for life in the community with skills – there are are Technical Instructors (T.I.s) and Occupational Therapists (O.T.s). Some of the skills I gained were travelling on buses in South East Northumberland, cooking, coping with crowds, finding a role (typically as a volunteer) and generally building a life outside of the hospital environment – all things that improve personal independence. Intellectual skills aren't a high priority in the NHS. Getting people stable, on the right medication and with the right amount of in the community support is a high priority.

For example, Contact, where I work as a volunteer , is a mental health charity based in Morpeth, Northumberland. [Contact] It was established in 1986 and it offers support and social contact for all its members. It has a computer project with two volunteers (me and Michael N) between the two of us we do all sorts of things. We provide front line support – looking after the PCs being used in the admin part of Contact, helping members of staff when a PC starts behaving in an unexpected manner, or tuition – helping members with computer issues – advice and guidance and help, dealing with infrastructure installation & maintenance issues (networks, phones, printers) installed and maintained by Michael N.

Lots of places have a free software policy. We (Contact) have a free hardware and free software philosophy – we take in unwanted PCs, fix and refurbish them and give them away to people with mental health problems, their carers or their children. As part of our philosophy we provide people with PCs with a selection of free/open source software. [STK]

As a volunteer, the work I do in Contact gives me similar benefits to the benefits other patients experience when working in the Kiff Kaff. It involves me with other people – good for preventing social withdrawal. It is also good for my morale to 'keep my hand in', making use of my degree in I.T. There are challenges – Contact's I.T. structure is balkanised. We have a variety of PCs with a variety of Windows and Linux versions. Officially the Crafts & Internet room is meant to be Windows based but Michael N and I are reluctant to move back to Windows – so we're using Ubuntu Linux in the Internet room and it works OK for us. Tristan S runs

The main foundation is simple: understanding and flexibility, structure and support.

the Kiff Kaff computers at the hospital and patients are always installing junk on his (Windows) PCs.

My work in Contact gives me cognitive challenges that other volunteer jobs don't give. When dealing with a newly donated computer, I have to work out why it was donated – was it too old, was it faulty, do we have a Windows licence for it? Some stuff gets broken down into parts to be used to fix other PCs. Some stuff is given to IM from the Tyneside Linux User Group for his various personal projects.

One of the hardest things is tracking down intermittent faults. I had a donated PC that booted nicely into Windows XP, had a reasonable amount of RAM with an AMD Athlon CPU. So I tried booting an Ubuntu 10.4 CDR and it failed. The screen just didn't look right. I downloaded copies of both memtest86 and memtest86+ and booted from both of them. At just over 10% complete, the screen display would become corrupted. Delving into the box revealed that the video card was wobbling a bit. So I dug out a spare video card, booted up and checked it out – it ran full memtests with no problems.

Another problem is that while I prefer to use Linux at home, I have to deal with a variety of Windows versions in Contact. So I sometimes have to fall back on the knowledge I picked up working on the WIN32 port of the LiBRiS search engine. If that doesn't work and checking for help via mailing lists (Tyneside LUG, accu-general) or Google doesn't work, I contact Michael N for suggestions.

Once the hardware is dealt with – first you fix it, then you memtest it, then you wipe the hard disk with dban (Darik's Boot and Nuke), then you install an operating system, then you install a *software toolkit* [STK]. My software toolkit is a collection of applications that routinely gets installed on PCs before they are given away. It include utilities, productivity applications and games. I've got a special form to help me keep track of the status of a PC being worked on. The act of working on these PCs helps exercise my brain.

There is a broad spectrum of employees – from volunteers to salaried employees. The volunteers would ideally be used on less demanding projects, with the benefit to the volunteer being the opportunity to gain current experience and learn new things. The path from volunteer to salaried employee is a broad one.

Intellectual activity and the NHS

Judging matters based on the sizeable population of patients I have seen, intellectual activity and its encouragement are rare. I was diagnosed late in life – roughly when I was 30.

If I'd been diagnosed when 18 years old, I would have been medicated and taken out of normal life and into the culture of being admitted onto and discharged from psychiatric wards at a key point in my development. Eventually I would be discharged into social housing (a council flat).

By being diagnosed later on in life, I experienced those things later on. I was able to do A levels, go to University and work as a programmer – something I am very grateful for. I was last an in-patient in hospital in 2004. I was suffering heavily from cognitive impairment and I used a variety of mental stimuli to return to a reasonable level of intellectual activity. Some teenage patients do go on to do academic work but all I can say on the subject is that personally it would have made a demanding situation even more difficult.

What can patients do?

Well... while care in the community and voluntary work are helpful, it means the I.T. tradition of moving house to get a new job just can't happen. So that means working as a volunteer (e.g. Contact), working for a software producing company and communicating over the internet, or in a FLOSS (Free/Libre Open Source Software) project (but which one?), writing book reviews for CVu. Some of these suggestions should ideally be supported by a mentor via email.

What advantages do patients have?

There is an evolutionary creativity bonus in favour of people with mental health problems. [Preti97]

Artistic endeavour is encouraged by the NHS – there is Art Therapy where the patients create art and it is analysed by an expert. Then there are certain aspects of Occupational Therapy – from creating Easter and Christmas cards to painting (pottery and woodwork have been discontinued).

Some illnesses are known to enhance mathematical ability. Certain articles in the press have discussed the benefits of having an autistic mind.

What can companies do?

The main foundation is simple: understanding and flexibility, structure and support. Companies would have to accept that sometimes people relapse and will eventually recover the ability that is usually impaired in such circumstances. However, when provided with supported opportunities/ projects for people to work on, people are more likely to stay well (stress permitting) for longer than those who don't do some form of work.

Flexibility in working practices would be necessary – one size does not fit all. Most people with mental health issues would typically be working for a much shorter working week than is conventional. Also, for lots of sufferers, travel is very difficult so a distance working opportunity would be good. As well as shorter hours, job sharing is a possible solution.

A key factor would be to provide work which *isn't* time critical. In Contact that could be refurbishing PCs, tutoring other members of Contact, providing initial technical support. In a software environment that would probably be R&D, testing and documentation. Some repetitive tasks can be a therapeutic activity – e.g. doing the washing up in the Kiff Kaff or installing free software packages (I've installed OpenOffice over 100 times now – on different PCs :).

What can the rest of us do?

Be accepting of people's foibles. Be accepting of people's disabilities. In particular, helping people stretch themselves into new roles without triggering a relapse. For example starting a new employee on a small project which acts as a refresher task for existing skills. Then suggest new

People in employment can experience mental health problems and providing support and flexible working conditions instead of making them redundant can be good for staff morale

things to learn, help with that learning and gradually build up a skill base that is custom made for your business.

What can companies gain?

Consider what companies can get from a socially responsible recruitment and staff management policy. People in employment can experience mental health problems and providing support and flexible working conditions instead of making them redundant can be good for staff morale. Having a public policy of supporting existing and new employees that have mental health problems can be a source of good public relations.

Companies can tap an otherwise untouched vein of creative talent, people with mental health problems are a good place to start. [BBC] ■

References and resources

[BBC] Autism sufferers in industry: http://news.bbc.co.uk/1/hi/8153564.stm

[Bruntlett] My blog: http://schizopanic.blogspot.com/

[Contact] Contact's web site: http://www.contactmorpeth.org.uk/

[C-Wiki] Contact's Wiki: http://contactmorpeth.wikispaces.com/

[NHS] NHS web site – http://www.nhs.uk/conditions/schizophrenia/Pages/Introduction.aspx

[Preti97] Preti, Antonio and Miotto, Paulo (1997) Creativity. Evolution and Mental Illnesses: http://cogprints.org/2009/1/preti_a%26miotto_p.html

[NewScientist] Advantages of autism: http://www.newscientist.com/article/mg20627581.500-the-advantages-of-autism.html

[NorthTypeWear] Northumberland, Tyne & Wear leaflets: http://www.ntw.nhs.uk/pic/leaflet.php?s=selfhelp

[StGeorge's] St Georges Hospital: http://www.stgeorgeshistory.org.uk/site/

[STK] Contact's software toolkit: http://contactmorpeth.wikispaces.com/SoftwareToolkit

[Times] Genetics and the link between maths and autism: http://www.timesonline.co.uk/tol/life_and_style/health/article2060584.ece

[Wikipedia] Wikipedia info: http://en.wikipedia.org/wiki/Schizophrenia

Acknowledgements

Thanks to Kevlin Henney, Ric Parkin and the Overload Editorial Readers for commenting on this article. Also thanks are due to accu-general for replies to a message about this topic. In particular special thanks are due Huw Lloyd for his help with the writing of this article.

Single-Threading: Back to the Future?

Dealing with multi-threading is notoriously hard. Sergey Ignatchenko learns lessons from the past.

So the 'multi-core revolution' is finally here [Merritt07, Suess07, Martin10] (some might argue that it has already been here for several years, but that's beyond the point now). Without arguing whether it is good or bad, we should agree that it is a reality which we cannot possibly change. The age of CPU frequency doubling every two years is long gone and we shouldn't expect any substantial frequency increases in the near future, and while there are still improvements in single-core performance unrelated to raw frequency increases, from now on we should count mostly on multiple cores to improve performance. Will it make development easier? Certainly not. Does it mean that everybody will need to deal with mutexes, atomics, in-memory transactions (both with optimistic and pessimistic locking), memory barriers, deadlocks, and the rest of the really scary multi-threading stuff, or switch to functional languages merely to deal with multi-threading? Not exactly, and we'll try to show it below. Please note that in this article we do not try to present anything substantially new, it is merely an analysis of existing (but unfortunately way too often overlooked) mechanisms and techniques.

How long is 'as long as possible'?

It is more or less commonly accepted that multi-threading is a thing which should be avoided for as long as it is possible. While writing multi-threaded code might not look too bad on the first glance, debugging it is a very different story. Because of the very nature of multi-threaded code, it is non-deterministic (i.e. its behavior can easily differ for every run), and as a result finding all the bugs in testing becomes unfeasible even if you know where to look and what exactly you want to test; in addition, code coverage metrics aren't too useful for evaluating coverage of possible race scenarios in multi-threaded code. To make things worse, even if the multi-threaded bug is reproducible, every time it will happen on a different iteration, so there is no way to restart the program and stop it a few steps before the bug; usually, step-by-step debugging offsets fragile race conditions, so it is rarely helpful for finding multi-threaded bugs. With post-mortem analysis of multi-threaded races using log files usually being impossible too, it leaves developer almost without any reliable means to debug multi-threaded code, making it more of a trial-and-error exercise based on 'gut feeling' without a real understanding what is really going on (until the bug is identified). Maintaining multi-threaded code is even worse: heavily multi-threaded code tends to be very rigid and fragile, and making any changes requires careful analysis and lots and lots of debugging, making any mix of frequently changed business logic with heavy multi-threading virtually suicidal (unless multi-threading and business logic are clearly separated into different levels of abstraction).

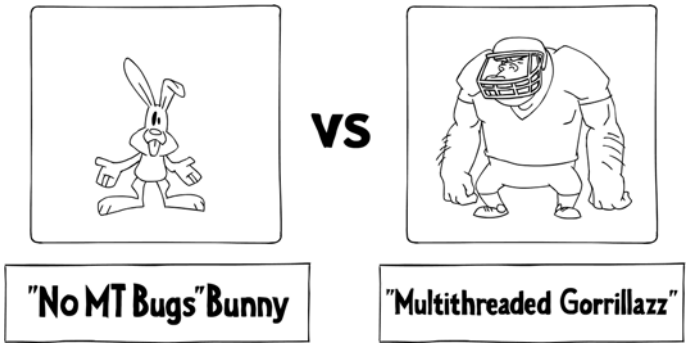
Sergey Ignatchenko has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

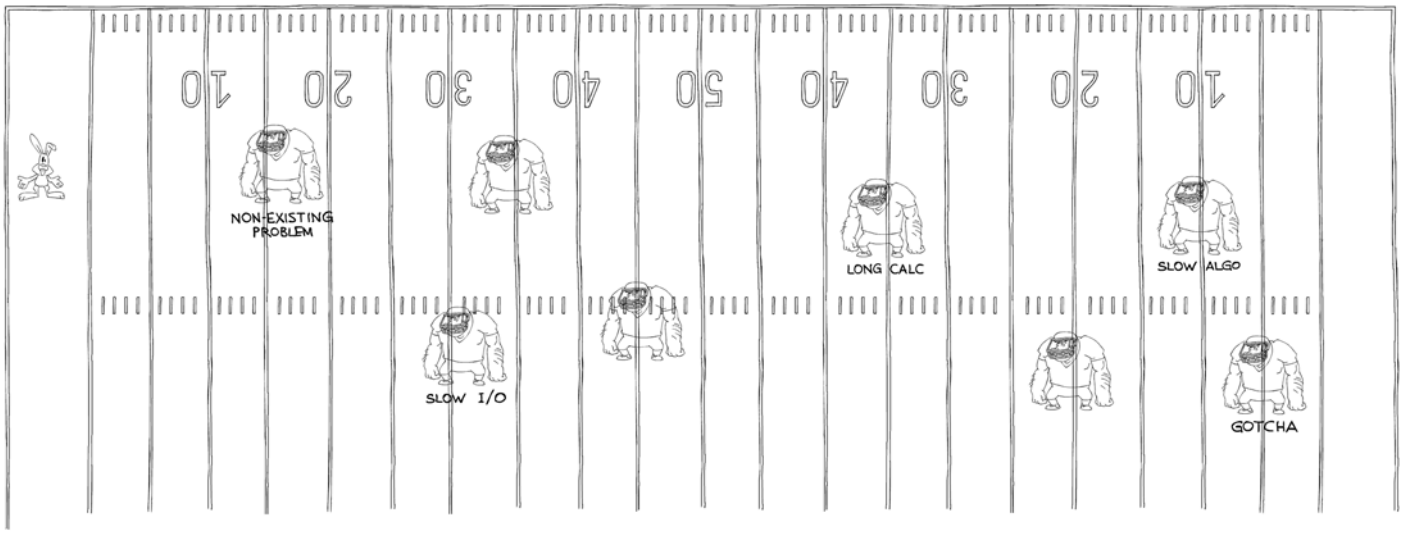
With all these (and many other) problems associated with multi-threaded code, it is easy to agree that multi-threading *should* be avoided. On the other hand, there is some disagreement on how long we *can* avoid it. In this article we will try to discuss how performance issues (if there are any) can be handled without going into too much detail of multi-threading. While not always possible, the number of cases when multi-threading can be avoided is extensive. And as discussed above whenever you can avoid it – you should avoid it, despite the potential fear that programs without multi-threading aren't 'cool' anymore. After all, the end-user (the guy who we all are working for) couldn't care less how many threads the program has or whether it utilizes all the available cores, as long as the program works correctly and is fast enough. In fact, using fewer cores is often beneficial for the end-user, so he's able to do something else at the same time; we also need to keep in mind that overhead incurred by multi-threading/multi-coring can be huge, and that Amdahl's Law provides only the *theoretical maximum* speedup from parallelization, with realized gains often being not even close to that. If a single-threaded program does something in a minute on one core, and multi-threaded one does the same thing in 55 seconds on 8 cores (which can easily happen if the granularity of context switching is suboptimal for any reason), it looks quite likely that user would prefer single-threaded program.

'No Multi-Threaded Bugs' Bunny & 'Multithreaded Gorrillazz'

Let us consider a developer who really hates dealing with those elusive multi-threading bugs. As he is our positive hero, we need to make him somewhat cute, so let's make him a bunny rabbit, with our hero becoming 'No Multi-Threaded Bugs' Bunny. And in opposition to him there is a whole bunch of reasons, which try to push him into heavy multi-threading with all the associated problems. Let's picture them as 'Multithreaded Gorrillazz' defending team. To win, our 'No MT Bugs' Bunny needs to rush through the whole field full of Gorrillazz, and score a touchdown. While it might seem hopeless, he has one advantage on his side: while extremely powerful and heavy, Gorrillazz are usually very slow, so in many cases he can escape them before they can reach him.

A few minor notes before the game starts: first of all, in this article we will address only programs which concentrate on interacting with the user one way or another (it can be a web, desktop, or mobile phone program, but





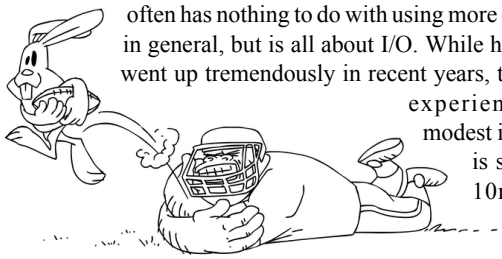
interaction should be a substantial part of it). Scientific calculations/HPC/video rendering farms/etc. is a completely different game which is played on a very different field, so we will not discuss it here. Second important note is that we're making a distinction between 'a bit of multi-threaded code in a very limited number of isolated places' and 'massive multi-threading all over the code'. While the former can usually be managed with a limited effort and (given that there are no better options) we'll consider it as acceptable, the latter is exactly the thing we're aiming to avoid.

Houston, have we got a problem?

So, our 'No MT Bugs' Bunny is standing all alone against the whole field of fierce Gorrillazz. What is the first move he shall make? First of all, we need to see if he's writing client-side code or server-side code. In the first two quarters of the game, we will concentrate on client-side code, with server-side considered in the quarters 3&4 (coming in the next issue). And if the application is client-side, then the very first question for our hero is the following: does his application *really* experience any performance problems (in other words, would users actually care if the application runs faster)? If not, he can simply ignore all the Gorrillazz at least at the moment and stay single-threaded. And while Moore's law doesn't work any more for frequencies, and modern CPUs got stuck at measly 3-4GHz, it is still about 1000 times more than the frequency of the first PCs, which (despite a 1000-times-less-than-measly-3-4-GHz-modern-CPUs speed) were indeed able to do a thing or three. It is especially true if Bunny's application is a business-like one, with logic like 'if user clicks button 'ok', close this dialog and open dialog 'ZZZ' with field 'XXX' set to the appropriate value from previous dialog'; in cases like the last one, it is virtually impossible to imagine how such logic (taken alone, without something such as associated MP4 playing in one of the dialogs – we'll deal with such a scenario a bit later), can possibly require more than one modern core regardless of code size of this logic.

To block or not to block – there is no question

If our 'No MT Bugs' Bunny does indeed have performance problems with his client application, then there is a big chance that it is related to I/O (if there are doubts, a profiler should be able to help clarify, although it's not always 100% obvious from the results of the profiling). If a user experiences delays while the CPU isn't loaded, the cause often has nothing to do with using more cores, or with CPU in general, but is all about I/O. While hard disk capacities went up tremendously in recent years, typical access time experienced much more modest improvements, and is still in the order of 10ms, or more than *ten to the seventh power* CPU clock cycles. No wonder, that if Bunny's program accesses the disk heavily the user can experience delays. Network access can incur even larger delays: while in a LAN typical round-trip time is normally within 1ms, typical transatlantic round-trip time is around 100-200ms, and that is only if everything on the way works perfectly; if something goes wrong, one can easily run into delays on the order of *seconds* (such as DNS timeout or a TCP retransmit), or even *minutes* (for example, the typical BGP convergence time [Maennel02]); the last number is about *eleven orders of magnitude* larger than the CPU clock time. As Garfield the Cat would have put it: 'Programmers who're blocking UI while waiting for network I/O, should be dragged out into the street and shot'.

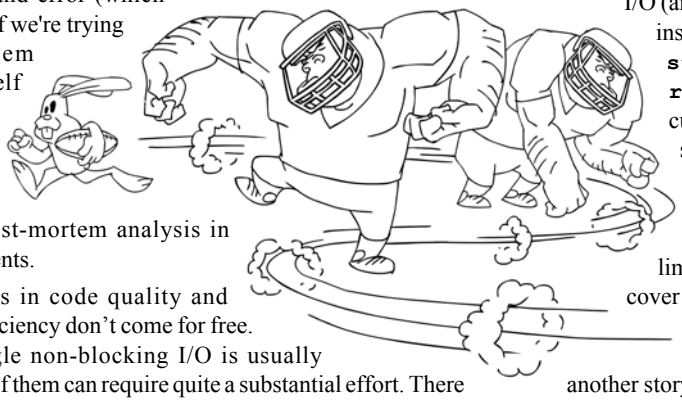


The way to avoid I/O delays for the user without going into multi-threading has been well-known since at least 1970s, but unfortunately is rarely used in practice; it is non-blocking I/O. The concept itself is very simple: instead of telling the system 'get me this piece of information and wait until it's here', say 'start getting this piece of information, return control to me right away and let me know when you're done'. These days non-blocking I/O is almost universally supported (even the originally 100%-thread-oriented Java eventually gave up and started to support it), and even if it is not supported for some specific and usually rather exotic API function (such as `FlushFileBuffers()` on Windows), it is usually not that big problem for our Bunny to implement it himself via threads created specially for this purpose. While implementing non-blocking I/O himself via threads will involve some multithreaded coding, it is normally not too complicated, and most importantly it is still confined to one single place, without the need to spread it over all the rest of the code.

Non-blocking I/O vs heavy multi-threading

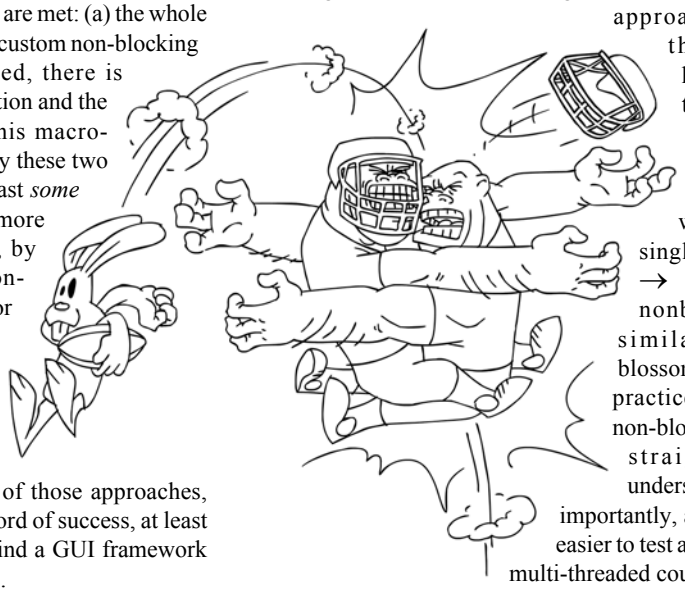
Unfortunately, doing things in parallel (using non-blocking I/O or via any other means) inherently has a few not-so-pleasant implications. In particular, it means that after the main thread has started I/O, an essentially new program state is created. It also means that program runs are not 100% deterministic anymore, and opens the potential for races (there can be differences in program behavior depending on at which point I/O has ended). Despite all of this, doing things in parallel within non-blocking I/O is still much more manageable than a generic heavily multi-threaded program with shared variables, mutexes etc. Compared to heavily multi-threaded approach, a program which is based on non-blocking I/O usually has fewer chances for races to occur, and step-by-step debugging has more chances to work. This happens because while there is *some* non-determinism in non-blocking I/O program, in this case a number of significantly different scenarios ('I/O has been completed earlier or later than certain other event') is usually orders of magnitude smaller than the potential number of different scenarios in a heavily multi-threaded program (where a context switch after *every* instruction can potentially cause substantially different scenarios and lead to a race). It can even be possible to perform a formal analysis of all substantially different scenarios due to different I/O timing, providing a theoretical proof of correctness (a

similar proof is definitely not feasible for any heavily multi-threaded program which is more complicated than 'Hello, World!'). But the most important advantage of non-blocking I/O approach is that, with proper level of logging, it is possible to reconstruct the exact sequence of events which has led to the problem, and to identify the bug based on this information. This means we still have some regular way to identify bugs, not relying on trial-and-error (which can easily take years if we're trying to identify a problem which manifests itself only in production, and only once in a while); in addition, it also means that we can also perform post-mortem analysis in production environments.



These improvements in code quality and debugging/testing efficiency don't come for free. While adding a single non-blocking I/O is usually simple, handling lots of them can require quite a substantial effort. There are two common ways of handling this complexity. One approach is event-driven programming, ubiquitous in the GUI programming world for user events; for non-blocking I/O it needs to be extended to include 'I/O has been completed' events. Another approach is to use finite state machines (which can vary in many aspects, including for example hierarchical state machines). We will not address the differences of these approaches here, instead mentioning that any such implementations will have all the debugging benefits described above.

One common problem for both approaches above is that if our Bunny has lots of small pieces of I/O, making all of them non-blocking can be quite tedious. For example, if his program makes a long search in a file then, while the whole operation can be very long, it will consist of many smaller pieces of I/O and handling all associated states will be quite a piece of work. It is often very tempting to separate a whole bunch of micro-I/Os into a single macro-operation to simplify coding. This approach often works pretty well, but only as long as two conditions are met: (a) the whole such operation is treated as similar to a kind of large custom non-blocking I/O; (b) until the macro-operation is completed, there is absolutely no interaction between this macro-operation and the main thread, except for the ability to cancel this macro-operation from the main thread. Fortunately, usually these two conditions can be met, but as soon as there is at least *some* interaction added, this approach won't work anymore and will need to be reconsidered (for example, by splitting this big macro-operation into two non-blocking I/O operations at the place of interaction, or by introducing some kind of message-based interaction between I/O operation and main thread; normally it is not too difficult, though if the interaction is extensive it can become rather tedious).



Still, despite all the associated complexities, one of those approaches, namely event-driven approach, has an excellent record of success, at least in GUI programming (it will be quite difficult to find a GUI framework which isn't event-driven at least to a certain extent).

If it walks like a duck, swims like a duck, and quacks like a duck...

If after escaping 'Long I/O' Gorrilla, Bunny's client-side program is still not working as fast as the user would like, then there are chances that it is indeed related to the lack of CPU power of a single core. Let's come back to our earlier example with business-like dialogs, but now let's assume that somewhere in one of the dialogs there is an MP4 playing (we won't ask why it's necessary, maybe because a manager has said it's cute, or maybe marketing has found it increases sales; our Bunny just needs to implement it). If Bunny would call a synchronous function `play_mp4()` at the point of creating the dialog, it would stop the program from going any further before the MP4 ends. To deal with the problem, he clearly needs some kind of asynchronous solution.

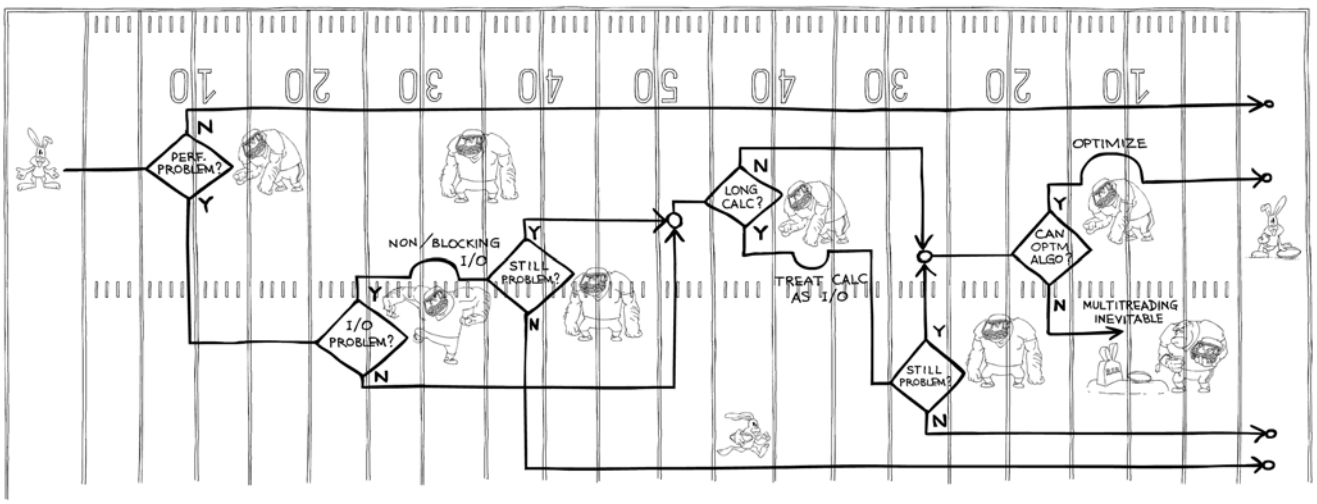
Let's think about it a bit. What we need is a way to start rendering, wait for it to end, and to be able to cancel it when necessary... Wait, but this is *exactly* what non-blocking I/O is all about! If so, what prevents our Bunny from representing this MP4 playback as a yet another kind of non-blocking I/O (and in fact, it **is** a non-blocking output, just using the screen instead of a file as an output device)? As soon as we can call `start_playing_mp4_and_notify_us_when_you_re_done()`, we can safely consider MP4 playback as a custom non-blocking I/O operation, just as our custom file-search operation we've discussed above. There might be a multi-threaded wrapper needed to wrap `play_mp4()` into a non-blocking API, but as it needs to be done only once: multi-threading still stays restricted to a very limited number of places. The very same approach will also cover lots of situations where heavy calculations are necessary within the client. How to optimize calculations (or MP4 playback) to split *themselves* over multiple cores is another story, and if our Bunny is writing yet another video codec, he still has more Gorrillazz to deal with (with chances remaining that one of them will get him).

Single-thread: back to the future

If our 'No MT Bugs' Bunny has managed to write a client-side program which relies on its main GUI thread as a main loop, and treats everything else as non-blocking I/O, he can afford to know absolutely nothing about threads, mutexes and other scary stuff, making the program from his point of view essentially a single-threaded program (while there might be threads in the background, our Bunny doesn't actually need to know about them to do his job). Some may argue that in 2010 going single-threaded might sound 'way too 1990-ish' (or even 'way too 1970-ish'). On the other hand, our single-thread with non-blocking I/O is not exactly the single-thread of linear programs of K&R times. Instead, we can consider it a result of taking into account the strengths and weaknesses of both previous approaches (classical single-threaded and classical heavily multi-threaded) and taking a small step further, trying to address the issues specific to both of them. In some very wide sense, we can even consider single-thread → multi-thread → single-thread-with-nonblocking-I/O transition, similar to Hegelian bud→ blossom→ fruit [Hegel1807]. In practice, architectures based on non-blocking I/O are usually more straightforward, can be understood more easily, and most importantly, are by orders of magnitude easier to test and debug than their heavily multi-threaded counterparts.

Last-second attempt

Our 'No MT Bugs' Bunny has already got far across the client side of the field, but if he hasn't scored his touchdown yet, he now faces the mightiest of remaining Gorrillazz, and unfortunately he has almost no room to maneuver. Still, there is one more chance for him to escape the horrible fate of heavily multi-threaded programming. It is good old algorithm optimization. While a speed up of a few percent might not be enough to keep you single threaded, certain major kinds of optimizations might make all the multi-threading (and multi-coring) unnecessary (unless, of course, you're afraid that a program without multi-core support won't look 'cool' anymore, regardless of its speed). If our Bunny's bottleneck is a bubble



sort on a 10M element array, or if he's looking for primes by checking every number N by dividing it by every number in 3..sqrt(N) range [Blair-Chappell10], there are significant chances that he doesn't really need any multi-coring, but just needs a better algorithm. Of course, *your* code obviously doesn't have any dreadfully inefficient stuff, but maybe it's still worth another look just to be 100% sure? What about that repeated linear scan of a million-element list? And when was the last time when you ran a profiler over your program?

Being gotcha-ed

Unfortunately, if our Bunny hasn't scored his touchdown yet, he's not too likely to score it anymore. He's been gotcha-ed by one of the Multithreaded Gorrillazz, and multi-threading seems inevitable for him. If such a situation arises, some developers may consider themselves lucky that they will need to write multi-threaded programs, some will hate the very thought of it; it is just a matter of personal preference. What is clear though is that (even if everything is done properly) it will be quite a piece of work, and more than a fair share of bugs to deal with.

Tools like OpenMP or TBB won't provide too much help in this regard: while they indeed make thread and mutex creation much easier and hide the details of inter-thread communication, it is not thread creation but thread synchronization which causes most of the problems with multi-threaded code; while OpenMP provides certain tools to help detecting race conditions a bit earlier, the resulting code will still remain very rigid and fragile, and will still be extremely difficult to test and debug, especially in production environments

Quarter 1&2 summary

While we have seen that our Bunny didn't score a touchdown every time, he still did pretty well. As we can see, he has scored 4 times, and has been gotcha-ed only once. The first half of the game has ended with a score of 'No Multi-Threaded Bugs' Bunny: 4 to 'Multithreaded Gorrillazz': 1. Stay tuned for remaining two quarters of this exciting match. ■

References

[Blair-Chappell10] How to become a parallel programming expert in 9 minutes, Stephen Blair-Chappell, *ACCU conference*, 2010

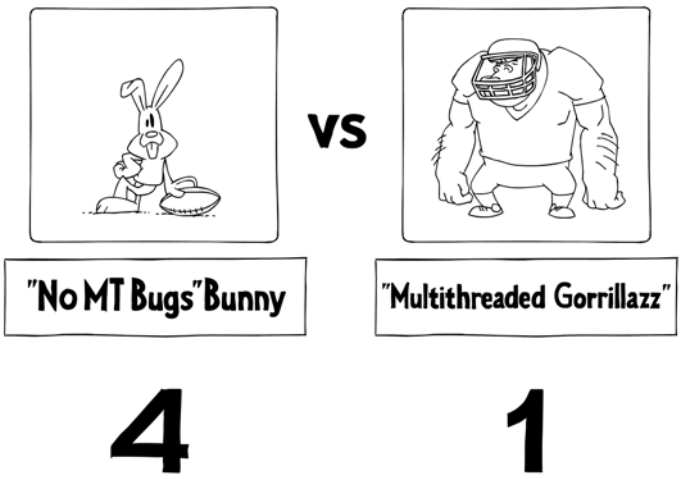
[Hegel1807] *Phenomenology of Spirit*, Hegel, 1807, translation by Terry Pinkard, 2008

[Maennel02] Realistic BGP traffic for test labs, Olaf Maennel, Anja Feldmann, *ACM SIGCOMM Computer Communication Review*, 2002

[Martin10] The Language Stew, Robert Martin, *ACCU Conference*, 2010

[Merritt07] M'soft: Parallel programming model 10 years off, Rick Merritt, 2007, http://www.eetimes.com/showArticle.jhtml?articleID=201200019

[Suess07] Is the Multi-Core Revolution a Hype?, Michael Suess, 2007, http://www.thinkingparallel.com/2007/08/21/is-the-multi-core-revolution-a-hype/



The Model Student: A Game of Six Integers (Part 3)

We now have the tools to analyse the Countdown Numbers Game. Richard Harris is ready to play.

In the first part of this article we described the numbers game from that Methuselah of TV quiz shows, Countdown [Countdown]. The rules of this game are that one of a pair of contestants chooses 6 numbers from a set of 4 large numbers and another of 20 small numbers, comprised respectively of the integers 25, 50, 75 and 100 and 2 of each of the integers from 1 to 10.

Over the course of 30 seconds both contestants attempt to discover a formula using the 4 arithmetic operations of addition, subtraction, multiplication and division and, no more than once, each of the 6 integers that results in a randomly generated target between 1 and 999 that has no non-integer intermediate values.

Reverse Polish Notation

Since we shall need to automatically generate formulae during any statistical analysis of the property of this game, we introduced Reverse Polish Notation, or RPN. Supremely well suited to computational processing, RPN places operators after the arguments to which they are applied, rather than between them as does the more familiar infix notation.

RPN utilises a stack to keep track of intermediate values during the processing of a formula and in doing so removes the need for parentheses to determine the order in which operators should be applied. Whenever a number appears in the input sequence it is pushed on to the stack and whenever an operator appears it pops its arguments off of the stack and pushes its result back on to it.

Formula templates

To simplify the enumeration of the set of possible formula, we introduced formula templates in which arguments are represented by *x* symbols and operators by *o* symbols. We described a scheme for recursively generating every possible formula template for up to a given number of arguments by repeatedly replacing *x* symbols with the sequence of symbols *xxo*.

The declaration of our C++ implementation of this scheme (the **`all_templates`** function) is:

```
std::set<std::string> all_templates(
   size_t arguments);
```

We concluded the first part of this article by implementing a mechanism with which we could evaluate formula template for a given set of operators and arguments.

We began by implementing an abstract base class to represent arbitrary RPN operators for a given type of argument, as illustrated in listing 1 together with the declarations of the 4 operator classes that are derived from it.

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

```
template<class T>
class rpn_operator
{
public:
  typedef
     std::stack<std::vector<T> >  stack_type;
  virtual ~rpn_operator();
  virtual bool apply(stack_type &stack) const = 0;
};
template<class T> class rpn_add;
template<class T> class rpn_subtract;
template<class T> class rpn_multiply;
template<class T> class rpn_divide;
```

Listing 1

```
template<class T>
class rpn_divide : public rpn_operator<T>
{
public:
  virtual bool apply(stack_type &stack) const;
};
```

Listing 2

Note that the return value of the **`apply`** member function indicates whether the result of the calculation has a valid value.

The implementation of the **`rpn_divide`** class is illustrated in listing 2. The remaining operators are implemented in much the same way, although divide is the only one that needs to check the validity, rather than just the availability, of its arguments and can therefore return **`false`** from its **`apply`** method. Specifically, it requires that for double arguments the second must not be equal to 0 and that for **`long`** arguments it must also wholly divide the first.

Note that the operator classes themselves are responsible for checking that the correct number of arguments are available, since this allows us to implement operators taking any number of arguments, should we so desire.

We then implemented the **`rpn_result`** structure to represent both the validity and the value of the result of a formula, as illustrated in listing 3.

```
template<class T>
struct rpn_result
{
  rpn_result();
  explicit rpn_result(const T &t);
  bool valid;
  T    value;
};
```

Listing 3

a permutation is the number of ways we can select a subset of elements from a set when order is important

```
template<class T>
rpn_result<T>
rpn_evaluate(const std::string &formula,
             const std::vector<rpn_operator<T>
                const *> &operators,
             const std::vector<T> &arguments);
```

Listing 4

Finally, we implemented a function that, given a **string** representing a formula template, a container of pointers to **rpn_operator** base classes and a container of arguments, yields the result of the formula generated by substituting the operators and arguments in sequence into the template. We plumped for the rather unimaginative name, **rpn_evaluate**, whose declaration is illustrated in listing 4.

Evaluating every formula for a given template

To examine the results of every possible formula in the Countdown numbers game we shall need to call the **rpn_evaluate** function for every one of the templates we have for up to 6 arguments with every possible set of operators and arguments. Recalling the formula we deduced for the total number of possible formulae

$$^{24}C_6 \times \sum_{i=1}^{6} T_i \times 4^{i-1} \times {}^{6}P_i$$

we concluded that we would need a mechanism of enumerating, for each n argument template, the 4^{n-1} choices of operators and the ${}^{6}P_n$ permutations of arguments in addition to a mechanism for enumerating the $^{24}C_6$ combinations of 6 from the 24 available numbers.

You will no doubt recall that a permutation is the number of ways we can select a subset of elements from a set when order is important and that a combination is the number of ways that we can select such a subset when order *isn't* important.

In the second part of this article we sought these mechanisms.

Fortunately for us, we created the first of them during our study of knots [Harris08]. The declaration of the **next_state** function that enumerates every possible state of a collection of integer-like objects is given below:

```
template<class BidIt, class T>
bool
next_state(BidIt first, BidIt last, const T &ub,
   const T &lb = T());
```

Since the **next_state** function is only applicable to iterator ranges of integer-like objects, we shall ultimately need to place our 4 operators in a container and use this function in conjunction with iterators ranging over it.

Generating the set of permutations was a little more complicated since the standard **next_permutation** function does not generate permutations of subsets. Fortunately, however, we were able to exploit the fact that it generates full sets of permutations in lexicographical order to trick it into generating permutations of subsets for us by reverse sorting the elements

```
template<class FwdIt> void
fill_iterator_vector(FwdIt first, FwdIt last,
                     std::vector<FwdIt> &vec);

template<class FwdIt, class T>
void
fill_dereference_vector(FwdIt first, FwdIt last,
                        std::vector<T> &vec);
```

Listing 5

from **mid** to **last**. Our **next_permutation** function enumerated the set of permutations of **mid-first** from **last-first** items in lexicographical order.

```
template<class BidIt>
bool
next_permutation(BidIt first, BidIt mid,
   BidIt last);
```

We then, rather tediously I suspect, devised our own algorithm for enumerating combinations in lexicographical order and finally, slightly less tediously I hope, implementing it. The declaration of the **next_combination** function in which we implemented our algorithm is provided below.

```
template<class BidIt>
bool
next_combination(BidIt first, BidIt mid,
                 BidIt last);
```

Putting it all together again

We concluded by implementing a function to iterate over every possible numbers game and pass their results to a statistics gathering function. We first needed some functions to convert between the sequences of values and iterators that our **rpn_evaluate** and argument and operator choice enumeration functions expected, as declared in listing 5.

We built the function to enumerate the possible games in two parts, the first of which iterated through the combinations of selections of numbers to work with, and the second of which iterated through every possible game that can be played with those numbers. The declarations of these two functions are illustrated in listing 6.

```
template<class BidIt, class Fun>
Fun
for_each_numbers_game(BidIt first, BidIt last,
   size_t args, Fun f);
template<class BidIt, class Fun>
Fun
for_each_numbers_game(BidIt first_number_choice,
   BidIt last_number_choice, Fun f);
```

Listing 6

a combination is the number of ways that we can select such a subset when order isn't important

```
class accumulator
{
public:
  accumulator();
  accumulator & operator+=(unsigned long n);
  operator double() const;

private:
  std::vector<unsigned long> n_;
};

accumulator::accumulator() : sum_(1, 0)
{
}

accumulator &
accumulator::operator+=(unsigned long n)
{
  assert(!sum_.empty());
  sum_.front() += n;
  if(sum_.front()<n)
  {
    std::vector<unsigned long>::iterator first =
       sum_.begin();
    std::vector<unsigned long>::iterator last  =
       sum_.end();
    while(++first!=last && ++*first==0);
    if(first==last)  sum_.push_back(1);
  }
  return *this;
}
```

Listing 7

We concluded that we were finally ready to begin analysing number games and we are indeed ready to do so.

Counting the number of valid games

Since the Countdown numbers game does not allow fractions, our formula for counting the number of formulae is going to overestimate the total number of valid games. I'm reasonably confident that I won't be able to derive an explicit formula to take this into account, so propose instead that we count them using our **`for_each_numbers_game`** functions.

Recall that our calculation implied that the number of formulae that could be constructed from 6 arguments chosen from 24 numbers was equal to 4,531,228,985,976, somewhat larger than can be represented by a 32 bit unsigned integer. We shall therefore recruit our **`accumulator`** class from our analysis of prime factorisations of integers [Harris09], given again in listing 7.

To check whether the result of an in-place addition requires additional storage we exploit the fact that in C++ unsigned integers don't overflow,

```
class
count_valid
{
public:
  void operator()(long);
  const accumulator & count() const;

private:
  accumulator count_;
};

void
count_valid::operator()(long)
{
  count_ += 1;
}

const accumulator &
count_valid::count() const
{
  return count_;
}
```

Listing 8

but that it instead treats arithmetic using *n* bit unsigned integers as being modulo 2^n [ANSI98].

We use the **`accumulator`** class in a function object that counts the number of valid formulae by adding 1 to the count every time it is called with the result of a calculation, as shown in listing 8.

Unfortunately, a preliminary investigation suggested that the calculation of the total number of valid games would take the best part of *2 months* of CPU time using the admittedly rather outdated PC upon which I am writing this essay.

Turbo-charging the stack

We can improve matters somewhat by abandoning the standard **`stack`** and implementing our own. This will exploit the short string optimisation to keep the bottom of the stack on the much faster local store. If your STL implementation already uses the short string optimisation for its **`vector`**s, then this won't really make much difference. Mine doesn't, so it will offer me an advantage, at least. The declaration of this class is given in listing 9.

Note that this isn't intended as a drop in replacement for the standard **`stack`**, since it only implements the member functions we require for an RPN calculation.

We use the trick of privately declaring, but not defining, the copy constructor and assignment operator to suppress the compiler generated defaults and ensure than an error will result if we accidentally use them. We don't ever need to copy stack objects during the evaluation of an RPN formula, and the defaults would leave them with pointers into each other's member arrays.

a preliminary investigation suggested that the calculation of the total number of valid games would take the best part of 2 months of CPU time

```
template<class T, size_t N,
   class Cont=std::deque<T> >
class rpn_stack
{
public:
  typedef T      value_type;
  typedef size_t size_type;

  rpn_stack();

  bool      empty() const;
  size_type size() const;

  const value_type & top() const;
  void push(const value_type& x);
  void pop();

private:
  rpn_stack(const rpn_stack &other);
  //not implemented
  rpn_stack & operator=( const rpn_stack &other);
  //not implemented

  std::stack<T, Cont> overflow_;
  value_type          data_[N];
  value_type *        top_;
};
```

Listing 9

We include a standard **stack** to cope with values that drop off the end of our member array. The implementation of the member functions is fairly straightforward, as illustrated in listing 10.

We are using the **data_** member array to store the first **N** values on the stack and so initialise the **top_** pointer to the start of it during construction. It shall always point to the element immediately following the last value on the stack that is stored in the member array. Only when we have exhausted the array will we use **overflow_**.

The **empty** member function therefore simply compares the **top_** pointer to the start of the member array.

Similarly, the **size** member function simply adds the number of values we currently have in our member array to the size of **overflow_**. If the former is full, the **top_** pointer will be equal to **data_+N** and so **size** will correctly return the size of **overflow_** plus **N**. If not, the latter will be empty and **size** will return the number of values currently in the member array.

The **top** member function defers to **overflow_** if it is not empty, otherwise returns the value immediately before **top_**. Note that, like the standard **stack**, we do not check that there are any values currently on the stack.

```
template<class T, size_t N, class Cont>
rpn_stack<T, N, Cont>::rpn_stack() : top_(data_)
{
}

template<class T, size_t N, class Cont>
bool
rpn_stack<T, N, Cont>::empty() const
{
  return top_==data_;
}

template<class T, size_t N, class Cont>
rpn_stack<T, N, Cont>::size_type
rpn_stack<T, N, Cont>::size() const
{
  return (top_-data_) + overflow_.size();
}

template<class T, size_t N, class Cont>
const rpn_stack<T, N, Cont>::value_type &
rpn_stack<T, N, Cont>::top() const
{
  return overflow_.empty() ? *(top_-1) :
     overflow_.top();
}

template<class T, size_t N, class Cont>
void
rpn_stack<T, N, Cont>::push(const value_type &x)
{
  if(top_!=data_+N)  *top_++ = x;
  else               overflow_.push(x);
}

template<class T, size_t N, class Cont>
void
rpn_stack<T, N, Cont>::pop()
{
  if(overflow_.empty())  --top_;
  else                   overflow_.pop();
}
```

Listing 10

The **push** member function assigns the pushed value to the element pointed to by **top_** and increments **top_** if it is not already at the end of the member array. If it is, the function simply defers to **overflow_**.

Similarly, if **overflow_** is empty, the **pop** member function simply decrements **top_**. If it is not, the function defers to it instead. Note that this function, like **top**, does not perform any check that there are any values currently on the stack.

what might happen to the ratio between valid integer-only formulae and total formulae as we increase the number of arguments

```
template<class T>
class rpn_operator
{
public:
  typedef rpn_stack<T, 6>  stack_type;
  ...
};
```

Listing 11

Finally, we should note that since the values stored in the member array are not destroyed until the `rpn_stack` is, it is not particularly useful for values that consume lots of resources.

Listing 11 shows the change that we need to make to the `rpn_operator` class to use the `rpn_stack`. Note that I'm only putting the first 6 values on the stack in local storage since during our analysis of the Countdown numbers game this will be the maximum number of arguments we shall use. Whilst this is clearly not a generally justifiable choice, I suspect that we'd be hard pushed to find another application for which we would really care quite this much about RPN formula calculation efficiency, so I figure it's probably OK.

Using our new stack brings the calculation time for counting every valid formula down to a little under 3 weeks. A fair bit better to be sure, but still not exactly tractable.

Whilst I'm sure that we could find a yet more efficient approach with a little more work, I rather suspect that it wouldn't consist of such satisfyingly independent constructs. That said, if there's some blindingly obvious improvement that doesn't muddle up the responsibilities of our various functions and classes, I'd be more than happy to hear about it.

The Countdown numbers game's little brothers

To keep the calculation manageable, I therefore suggest that we instead consider a cut down version of the numbers game; one in which we choose our 6 arguments from the set of 4 large numbers and half of the set of small numbers, or in other words 1 of each of the integers from 1 to 10.

Another back of the envelope calculation suggests that this should take approximately 10 hours; hardly nippy, but not completely out of the question.

Multiplying the number of formulae we can construct with up to 6 arguments by the number of ways in which we can select 6 arguments from 14 without taking order into account, or ${}^{14}C_6$, yields a total of 101,097,214,218 possible formulae, still a smidge out of the range of a 32 bit unsigned integer.

The result of our calculation for this smaller numbers game shows that there are just 32,215,124,261 valid formulae, less than a third of the total number of formulae. The 68,882,089,957 invalid formulae must all involve division since that's the only operator that can have an invalid result. We can easily count the number of formulae involving division operations by subtracting from our total the number of formulae that involve only addition, subtraction and multiplication (we calculate this by replacing the 4 with a 3 in our formula). Doing so indicates that there are exactly 76,425,599,250 formulae involving division, of which over 90% are invalid.

This leads me to wonder what might happen to the ratio between valid integer-only formulae and total formulae as we increase the number of arguments. This calculation will be more expensive still, so we shall examine games using 1 of each of the integers from 1 to 8 with from 1 to 8 arguments. The results of this calculation are given in figure 1.

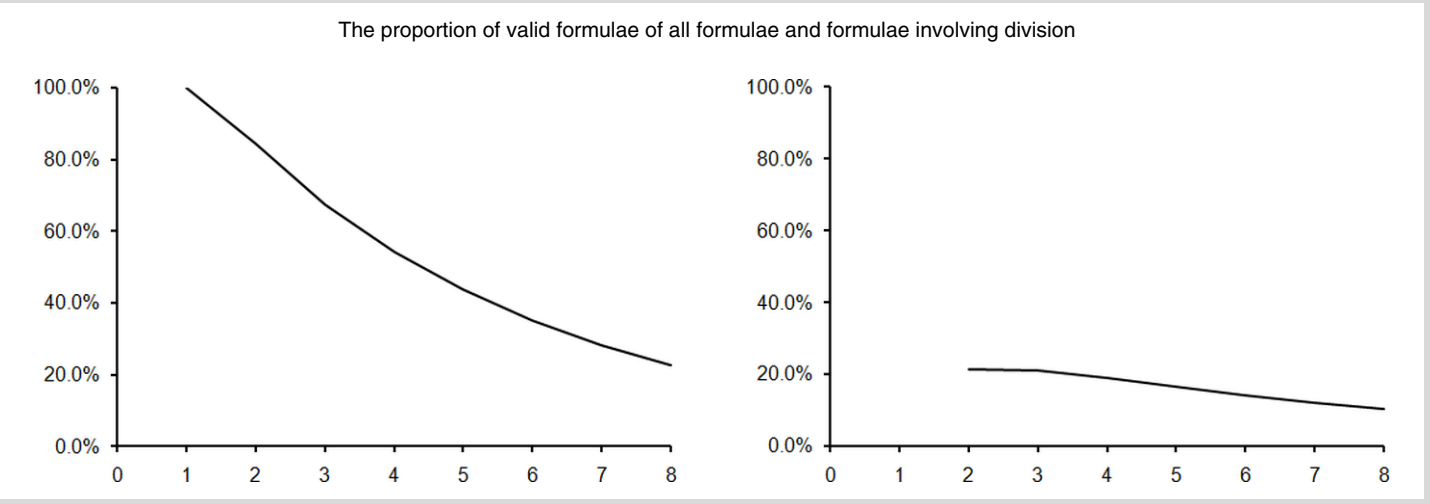


Figure 1

As fascinating as this almost is, it's high time we got around to investigating the statistical properties of the game

```
class game_histogram
{
public:
  game_histogram();
  game_histogram(long lower_bound,
                 long upper_bound);

  long   lower_bound() const;
  long   upper_bound() const;
  double samples() const;

  double operator[](long i) const;
  void   operator()(long i);

private:
  typedef accumulator             value_type;
  typedef std::vector<value_type> histogram_type;

  long           lower_bound_;
  value_type     samples_;
  histogram_type histogram_;
};
```

Listing 12

As fascinating as this almost is, it's high time we got around to investigating the statistical properties of the game. Specifically, I should like to know what the distribution of the results of every valid numbers game looks like.

Building a histogram of numbers game results

Since the results of the numbers game formulae are integers, their distribution will be discrete and hence naturally represented with a histogram. Listing 12 provides the declaration of the **`histogram`** class we shall use.

It is a little less fully featured than the histograms we have implemented for previous studies, although since it uses our **`accumulator`** class to keep count of the samples it can deal with much larger numbers.

We can afford a reduced interface since the sample values will be integers and will thus serve perfectly well as indices into the histogram. Note that the function call operator overload is used to add samples since this enables us to use the **`game_histogram`** class as the function object expected by our **`for_each_numbers_game`** function.

Listing 13 illustrates the definitions of the member functions. These are reasonably straightforward, with the only real gotcha being that the **`lower_bound`** and **`upper_bound`** are inclusive bounds of the histogram.

When trying to read values outside of the stored range we simply return 0, and when trying to add them the histogram entry is quietly dropped and only the count of the number of samples is increased.

```
game_histogram::game_histogram() :
lower_bound_(0)
{
}

game_histogram::game_histogram(long lower_bound,
                               long upper_bound) :
         lower_bound_(std::min(lower_bound,
                               upper_bound)),
         histogram_(size_t(labs(
                     upper_bound-lower_bound)+1))
{
}

long
game_histogram::lower_bound() const
{
  return lower_bound_;
}

long
game_histogram::upper_bound() const
{
  return lower_bound_ + (
     long(histogram_.size()) - 1);
}

double
game_histogram::samples() const
{
  return samples_;
}

double
game_histogram::operator[](long i) const
{
  if(i<lower_bound() || i>upper_bound())
     return 0.0;
  return histogram_[size_t(i-lower_bound_)];
}

void
game_histogram::operator()(long i)
{
  samples_ += 1;
  if(i>=lower_bound() && i<=upper_bound())
  {
    histogram_[size_t(i-lower_bound_)] += 1;
  }
}
```

Listing 13

I must admit that I'm a little surprised by the shape of the histogram

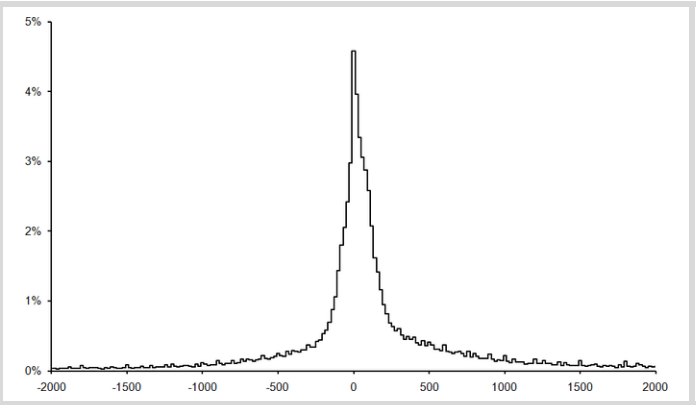


Figure 2

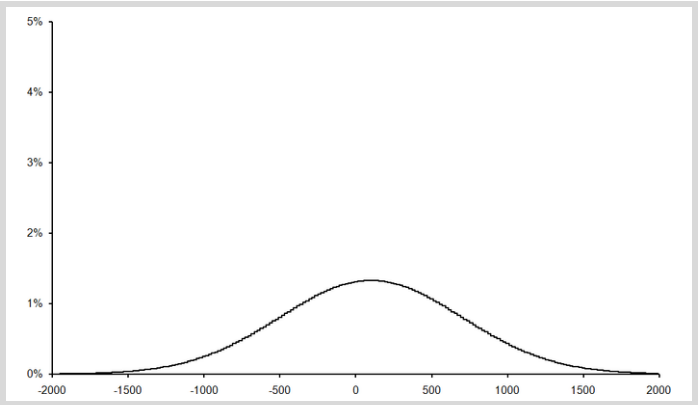


Figure 3

The histogram generated with this class enumerated over every game with the 4 large numbers and the 10 small numbers with results ranging from -2000 to +2000, grouped into buckets of 20 results to smooth out the graph a little, is shown in figure 2.

There are clearly more positive results than negative, which should be expected since we can only generate a negative result by subtracting some formula from 1 of the 6 selected numbers, leaving that formula with only 5 or fewer arguments.

It is worth noting that the formulae with results in the range of our histogram account for just a little over 70% of the valid games.

I must admit that I'm a little surprised by the shape of the histogram. I was expecting it to look like a discretisation of the normal distribution since that, as you will remember from our previous studies, is the statistical distribution of sums of random numbers.

Any formula can be recast as a sum of terms involving just multiplication and division by expanding out any terms in brackets, something known as the distributive law of arithmetic. For example, we can expand the formula

$$a \times (b + c)$$

into

$$(a \times b) + (a \times c)$$

With an admittedly rather hand-waving argument I had imagined that these terms could be thought of as random variables in their own right. Whilst the terms would clearly not be independent in general, I believed that assuming that they were would yield a reasonable approximation.

The average, or mean, of our truncated histogram is approximately 100, whilst its standard deviation (the square root of the average squared difference from the mean) is approximately 600. The histogram that would result from a normal distribution with these parameters is shown in figure 3 for comparison with our histogram of numbers game results.

Clearly I was very much mistaken.

So what kind of distribution do the results of the numbers games exhibit?

The histogram of the absolute values of numbers game results

In pursuit of a statistical distribution that might describe the numbers game, I suggest that we allow the result of a formula to be negated. There are trivially twice as many formulae that can be constructed and the distribution of the results must be symmetric about 0. We can therefore simply study the histogram of the absolute results (i.e. ignoring the sign) which we can build by adding together the histogram value for each positive result to the value for its negation, and from which we can easily construct the full distribution.

Figure 4 illustrates the histogram of the absolute results, also constructed with buckets of 20 results.

The fact that this distribution falls away to 0 so slowly is highly suggestive of the class of distributions it falls into; the power law distributions.

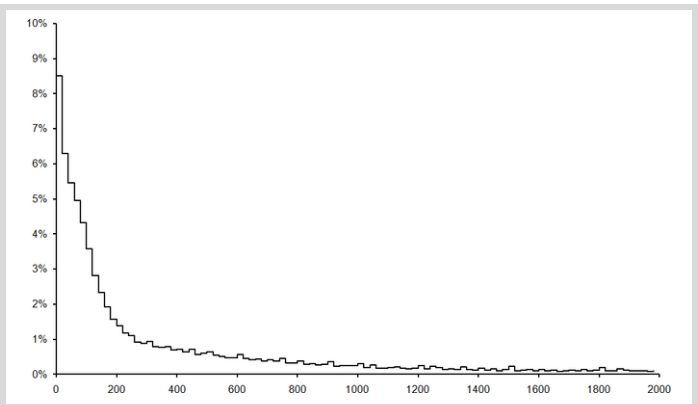


Figure 4

we cannot swiftly enumerate every possible formula in the full Countdown numbers game

Power law distributions

Power law distributions have the probability density functions of the form

$$p(x) = L(x) \times x^{-\alpha}$$

where L is a slowly varying function, or in other words has the limiting behaviour

$$\lim_{x \to \infty} \frac{L(tx)}{L(x)} = 1$$

for constant t [Clauset09]. The lim term on the left hand side of the fraction stands for the limit as x grows ever larger of the term to its right. We can interpret the equation as meaning that the function L should get closer and closer to a constant as x grows larger and larger.

Probability density functions, or PDFs, are identical to histograms when describing discrete random variables. PDFs are more general, however, in that they can also describe continuous random variables.

Power law distributions are notable because their PDFs decay to 0 very slowly as x increases and are hence associated with processes which display extreme behaviour with surprisingly high probability. The stock market, with its relatively frequent and all too hastily dismissed crashes, is one example that depressingly springs to mind [Ormerod05].

To determine whether a variable has a power law distribution, we can plot the logarithm of its PDF against the logarithm of x. If it follows a power law distribution then for large x this graph will be close to a straight line since

$$\begin{aligned}\lim_{x \to \infty} \ln p(x) &= \lim_{x \to \infty} \ln(L(x) \times x^{-\alpha}) \\ &= \lim_{x \to \infty} \ln L(x) - \lim_{x \to \infty} \alpha \ln x \\ &= c - \alpha \ln x\end{aligned}$$

for some constant c.

For sample data, in which there will inevitably be some loss of information, be it noise or gaps in the distribution, we instead sort the n samples in ascending order and, treating the first as having an index of 0, plot $(n\text{-}i)/n$ against the i^{th} value. This is the discrete equivalent of another test based on the integral of the PDF.

Unfortunately, neither of these tests is particularly accurate. Worse still, accurately determining whether a sample is consistent with a power law distribution is something of an open question with the most reliable current techniques relying upon barrages statistical tests that are far beyond the scope of this article.

Are the absolute numbers games power law distributed?

Our distribution seems to have a lot of little spikes in its tail and furthermore has an upper bound of

$$100 \times 75 \times 50 \times 25 \times 10 \times 9 = 843,750,000$$

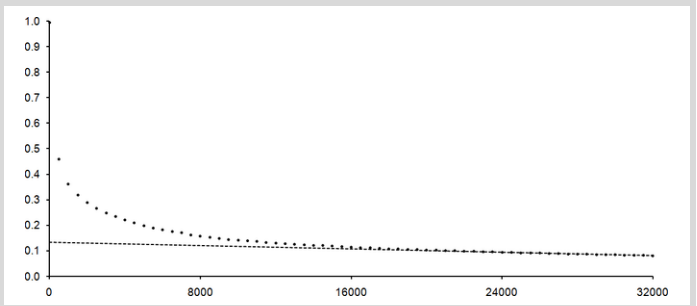


Figure 5

beyond which the probability of observing a result is trivially 0. It is therefore certainly not exactly power law distributed, although it is possible that it follows one approximately.

Using the sample data approach of identifying power law behaviour figure 5 plots every 500$^{\text{th}}$ of the sorted results from 0 to 32,000 against the function of their positions in the list together with a straight line drawn through every 500$^{\text{th}}$ of the 24,000$^{\text{th}}$ to the 32,000$^{\text{th}}$ point.

Well, it certainly *looks* very much like a straight line, and with a root mean square error between the line and the points it was drawn through of a little over 0.00025 we can probably conclude that it is at least quite close to one.

Despite this not being a particularly good test for power law behaviour, the cumulative histogram of the numbers game can be approximated for every result in this range by that of a power law with a root mean square error of approximately 0.0003. Statistically speaking, this is actually a fairly significant difference since the histogram is constructed from a huge number of results. Nevertheless, as an approximation it predicts the proportion of results falling below a given value within this range with an error that is everywhere less than a not too shabby 0.061%.

Since we cannot swiftly enumerate every possible formula in the full Countdown numbers game, we shall instead have to randomly sample them if we wish to check whether or not their absolute results are similarly approximated by a power law distribution.

Sampling the Countdown numbers game

To implement a scheme for randomly sampling numbers games, we take similar steps to those we took when implementing a function to enumerate them. Specifically we need a mechanism for the random generation of permutations, of combinations and of operator choices.

Just as our previously implemented `next_state` function solved the problem of enumerating operator choices, so the `random_state` function implemented as part of the same article solves the problem of randomly generating them. Implemented in listing 14 together with a random number generator, `rnd`, it was inspired by the standard `random_shuffle` function and, like it, does not place many restrictions upon the values on which it operates. This means that we shall not need

why did we spend so much time mucking about with combinations when permutations seem to do the job perfectly adequately

```
double
rnd(double x)
{
  return x * double(
     rand())/(double(RAND_MAX)+1.0);
}

template<class FwdIt, class States>
void
random_state(FwdIt first, FwdIt last,
             const States &states)
{
  while(first!=last)
  {
    *first++ = states[size_t(
       rnd(double(states.size())))];
  }
}
```

Listing 14

to use iterators into a container of operators, but will instead be able to use the container directly.

To generate a random permutation, we can exploit the fact that the standard **`random_shuffle`** function has an equal probability of leaving any value in any position. We can therefore simply use it and then just examine the part of the iterator range that we're interested in.

If we then sort that part of the iterator range, we have effectively implemented an algorithm for generating random combinations.

Our scheme shall therefore proceed as follows:

1. Pick a formula template at random.
2. Pick a set of operators at random.
3. Pick a combination of the available numbers at random.
4. Pick a permutation of arguments from these at random.

Hang on a sec, those last 2 steps look a bit dodgy to me.

We're picking a sorted random combination of numbers and then an unsorted random permutation of arguments from them. Our algorithms for generating random combinations and permutations require that we apply **`random_shuffle`** to the whole range from which each is generated and then ignore those elements we're not interested in. In other words, we shall be sorting the selected numbers to create the combination and then immediately afterwards randomly shuffling them again to generate our permutation of arguments. Isn't this ever so slightly a complete and utter waste of time?

Well, yes it is; we can combine both steps by instead generating a random permutation of the arguments we require from the full set of available numbers:

1. Pick a formula template at random.
2. Pick a set of operators at random.
3. Pick a permutation of arguments from the available numbers at random.

```
1 2 3 | 4 5 6
1 2 3 | 4 6 5
1 2 3 | 5 4 6
1 2 3 | 5 6 4
1 2 3 | 6 4 5
1 2 3 | 6 5 4
```

Figure 6

So why did we spend so much time mucking about with combinations when permutations seem to do the job perfectly adequately?

The answer lies in a subtle difference between the mechanics of sampling and those of enumeration. For a given formula template and set of operators, enumerating the permutations of 6 numbers from the 24 available will result in us counting some functions many times over. Specifically, for templates with less than 5 arguments there will be multiple permutations for which those arguments will be identical, as illustrated in figure 6.

This isn't an issue when sampling since the order of the unused numbers has no bearing whatsoever on the result of the formula. Since each ordering has exactly the same probability, the statistical distribution of the results is unaffected.

That settled, the function for randomly sampling numbers games is given in listing 15. This is a relatively straightforward implementation of our algorithm; we first randomly select a formula template, we then randomly generate a correctly sized vector of operators to substitute into it and we finally generate a random permutation of arguments. Note that we still have to copy the arguments into an appropriately sized **`vector`** since this is what our **`rpn_evaluate`** function expects. Figure 7 gives the power law graph of a sample of 40,000,000,000, or roughly 1%, of the Countdown numbers games. Once again, it looks like it tends to a straight line and the root mean square error between the line and the graph from the 24,000th to the 32,000th point of approximately 0.00035 supports this.

A theoreticalish justification

Mucking about with graphs and tests is all well and good, but it doesn't really provide any insight into the statistical behaviour of numbers games. What we really need is a theoretical justification as to why power law distributions might be reasonable approximations of them.

We shall begin by introducing yet another numbers game. In this new game we shall begin by picking, at random, a real number between 1 and 2. We shall then toss a coin; if it comes up tails the games is over and the number we picked is the result and if it comes up heads we double the number and toss the coin again, treating the doubled result as our starting point.

What we really need is a theoretical justification as to why power law distributions might be reasonable approximations of them

```
template<class Fun>
Fun
sample_numbers_games(size_t samples,
                     std::vector<long> numbers,
                     size_t args,    Fun f)
{
  typedef rpn_operator<long>
     const * operator_type;
  typedef std::vector<operator_type>
     operators_type;
  typedef std::vector<long> arguments_type;

  if(args>numbers.size())
     throw std::invalid_argument("");

  operators_type operators(4);
  const rpn_add<long> add;
     operators[0] = &add;
  const rpn_subtract<long> subtract;
     operators[1] = &subtract;
  const rpn_multiply<long> multiply;
     operators[2] = &multiply;
  const rpn_divide<long> divide;
     operators[3] = &divide;
  const std::set<std::string>
     templates(all_templates(args));
  operators_type used_operators;
  arguments_type used_arguments;

  while(samples--)
  {
    std::set<std::string>::const_iterator t
       = templates.begin();
    std::advance(t,
       size_t(rnd(templates.size())));
    const size_t t_args = (t->size()+1)/2;
    used_operators.resize(t_args-1);
    random_state(used_operators.begin(),
       used_operators.end(), operators);

    std::random_shuffle(numbers.begin(),
       numbers.end());
    used_arguments.assign(numbers.begin(),
       numbers.begin()+t_args);
    const rpn_result<long> result
       = rpn_evaluate(*t, used_operators,
                      used_arguments);
    if(result.valid)  f(result.value);
  }
  return f;
}
```

Listing 15

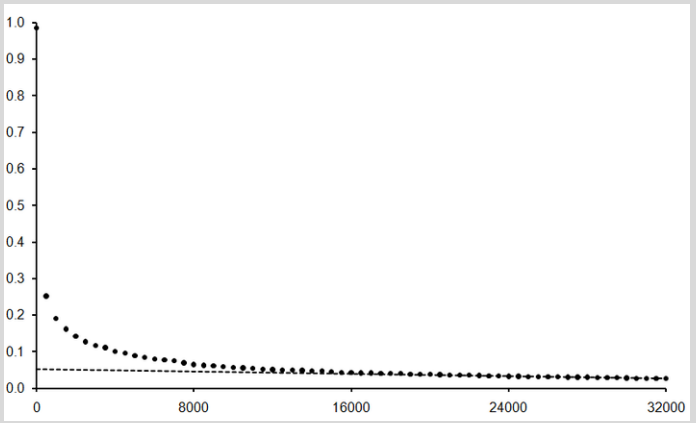


Figure 7

This game has a 1 in 2 chance of a result between 1 and 2, a 1 in 4 chance of a result between 2 and 4, a 1 in 8 chance of a result between 4 and 8 and so on. The probability density function of the results of this game, being the continuous limit of a histogram and for which the area under the curve between 2 values gives the probability that a result in that range will be observed, is given in figure 8. This curve is bounded by the inequality

$$\frac{1}{2x^2} \le p(x) \le \frac{2}{x^2}$$

as is illustrated by the dotted lines.

Clearly this is approximately an inverse square power law distribution. This fact is demonstrated even more clearly by the cumulative distribution function; the function that yields the area under the curve between 0 and any given value and hence the probability that we should observe a result less than or equal to that value, as illustrated in figure 9. The dotted line

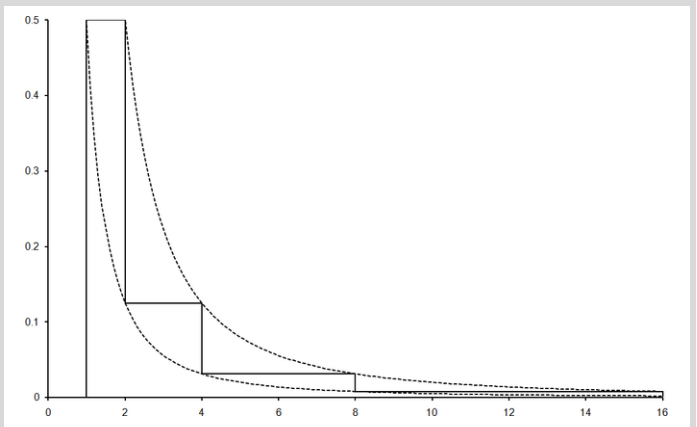


Figure 8

there is therefore a surprising relationship between the Countdown numbers game and the recent economic meltdown

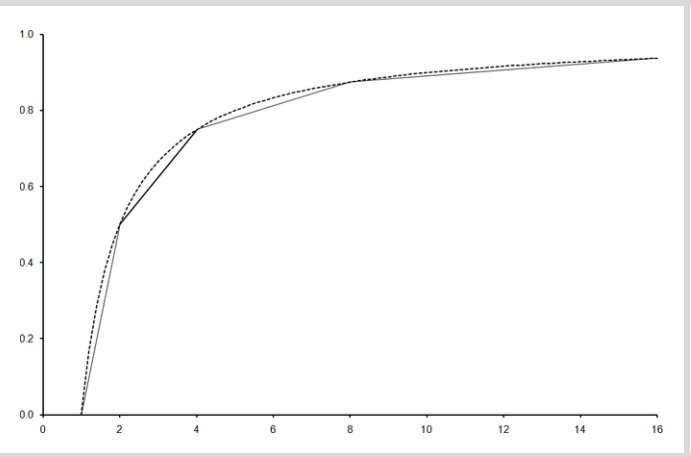


Figure 9

in this graph is the cumulative distribution function of a quantity exactly obeying an inverse square power law distribution.

This result can be generalised in that a game in which we pick a real number between 1 and n and with probability p choose to multiply the current result by n rather than quit the game must also be similarly approximated by a power law distribution. Furthermore, we needn't specify which statistical distribution governs the choice of the first number for this to hold.

We can identify the first number as the relatively small result of a formula involving the addition and multiplication of relatively small numbers in the Countdown numbers game. The successive multiplications can be similarly identified with multiplying these small results by the large numbers.

Note that for each formula with n arguments, there is a formula with n+1 arguments that equates to multiplying the former by the final argument. The proportion of such formulae to all formulae with n+1 arguments is equal to

$$\begin{aligned}\frac{1}{4}\times\frac{T_n}{T_{n+1}} &= \frac{1}{4}\times\frac{{}^{2n-1}C_n}{2n-1}\times\frac{2n+1}{{}^{2n+1}C_{n+1}}\\ &= \frac{1}{4}\times\frac{(2n-2)!}{n!(n-1)!}\times\frac{n!(n+1)!}{(2n)!}\\ &= \frac{1}{4}\times\frac{(n+1)\times n}{2n\times(2n-1)}\\ &= \frac{n+1}{16n-8}\end{aligned}$$

For sufficiently large n this is approximately constant, so the probability of multiplying an n argument formula by a large number is also approximately constant.

Subsequent divisions, should they be valid, can be thought of as reducing the probability of a multiplication, and subsequent additions will affect the larger results by orders of magnitude less than the multiplications.

It doesn't seem entirely unreasonable therefore to accept this latest game as an approximate model for the Countdown numbers game and that the latter can therefore be well approximated by a power law distribution.

Whilst the game is certainly not exactly governed by a power law, I believe that we can conclude with some confidence that we can reasonably approximate it with one and that there is therefore a surprising relationship between the Countdown numbers game and the recent economic meltdown.

Who'd have thunk it? ■

Acknowledgements

With thanks to Keith Garbutt for proof reading this article.

References and further reading

[ANSI98] *The C++ Standard*, American National Standards Institute, 1998.

[Clauset09] Clauset, A. *et al*, Power Law Distributions in Empirical Data, arXiv:0706.1062v2, www.arxiv.org, 2009.

[Countdown] http://www.channel4.com/programmes/countdown

[Harris08] Harris, R., The Model Student: A Knotty Problem, Part 1, *Overload* 84, 2008.

[Harris09] Harris, R., The Model Student: A Primal Skyline, Part 2, *Overload* 93, 2009.

[Ormerod05] Ormerod, P., *Why Most Things Fail*, Faber and Faber, 2005.

The Functional Student: A Game of Six Integers

The Countdown numbers game is a popular challenge. Richard Polton tries a new language to solve it.

When I saw Richard Harris's 'Model Student – A Game Of Six Integers' in *Overload 95* [Harris] I thought to myself that this was just crying out for a quick piece of functional programming and, as luck would have it, I had just recently secreted an installation of F# [Microsoft] on my PC between Visio and Powerpoint and was both researching the language and refreshing my mind as to that programming approach.

My first thought for the implementation was to attempt the solution as a logical statement, in the prolog sense, but after reflection I decided not to proceed in that direction because I wanted to be able to show close solutions as well as exact solutions, and I didn't feel I had the time to work out how to implement that in this new-to-me language, F#. Next I thought about a non-binary tree walk, where each of the six possible values for the first number occupied the first layer, then the four operators were four branches from each node, then the five possible numbers in the next layer, then the four operators, etc., etc. This, however, seemed far too messy and, consequently, I realised that I was still thinking about this in terms of the data instead of in terms of the solution process.

Thus the code in Listing 1 (overleaf) was born. The problem has, essentially, two sets of variables; six integers `intN`; and five operators `opN`, where each of the five operators can take one of four values, '+', '`-`', '`*`' or '`/`'. I started with the statement of the problem, ie (((((int1 op1 int2) op2 int3) op3 int4) op4 int5) op5 int6)-desiredResult=0, and rearranged it into, more or less, what you see encoded in the countdown function. This function is slightly more complex than above because it maintains a text string describing the path taken simultaneously with the actual calculation.

The core function in this programme is the `op` function. This takes two parameters; the first is the running total and the second is the next integer to be considered. That is, after the tuples have been 'opened up', `hd1` contains the string showing the path taken to reach the running total, which is itself contained within `hd2`. `y2` is the next integer in the sequence to be processed. Each of the four possible operators is applied and this leads to four outputs for each input, all returned in a single list. This list shows all the possible operator combinations for a given ordered sequence of integers. For example, `op [1] 2` returns `[ 1+2; 1-2; 1*2; 1/2 ]` except that `1/2` is discarded because it is not integral. Note that I have considered negative numbers as allowable but, if desired, they could be excluded easily using the same mechanism as after the division.

To utilise the `op` function, I needed a function to calculate the permutations of a sequence of numbers. I found something on `stackoverflow` [StackOverflow] which I could modify to do exactly what I wanted. The `calcPermutations` function determines all the permutations of a list of integers and returns a list of lists of the possibilities, eg `calcPermutations [1;2]` returns `[ [1;2] ; [2;1] ]`. This list of lists is then fed into a `lambda` function whose purpose is to process each of the sub-lists in turn, naming the individual components (the permuted integers) of each sub-list so that they can be used in the `op` function. At which point it becomes clear that Bob is, as they say, your mother's brother. All that remained now is to filter the resulting list of lists into those lists that satisfy the matching requirement and discard the rest.

Easy! At least, looking at it now, that is how it seems, but believe me, when you're buried beneath tons of virtual paper it quickly becomes a tangled mess of mind-melting code. Still, I managed to push the paper (to one side), dedicated a couple of hours of thinking time and lo! the result.

So, to run it the parameters to countdown are the six integers and the sought result is, unsurprisingly, `desiredResult`.

Note that I have not proven this works, only shown that it appears to work for the example case, as my day is filled with all those allegedly important tasks that I do in order to keep my team writing code.

Back to the future

Climbing out of my DeLorean [DeLorean], I can tell you that this programme will be extended in a number of interesting ways, not least being a trivial enhancement to use an increased set of operators. All that is required is an additional line for each new operator in the `op` function. For example, should we decide to consider 'remainder' as a valid operator in this context then we could extend the result list in `op` to include the tuple `(String.Format("{0}%{1}",hd1,y2),hd2%y2)`. Similarly, I have forseen a change to the code such that it accepts an arbitrary number of integers. As the code stands, it is hard-coded to accept six integers. Also, I have discerned that the programme will be extended to accept data types other than integers as parameters, for example strings, which could be combined using string-related operators. Not wanting to spoil the surprise, however, I shall avoid revealing any more for the present. ■

References

[DeLorean] http://www.delorean.com/

[Harris] 'The Model Student: A Game Of Six Integers (Part 1)', *Overload* 95, http://accu.org/index.php/journals/1607

[Microsoft] http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/

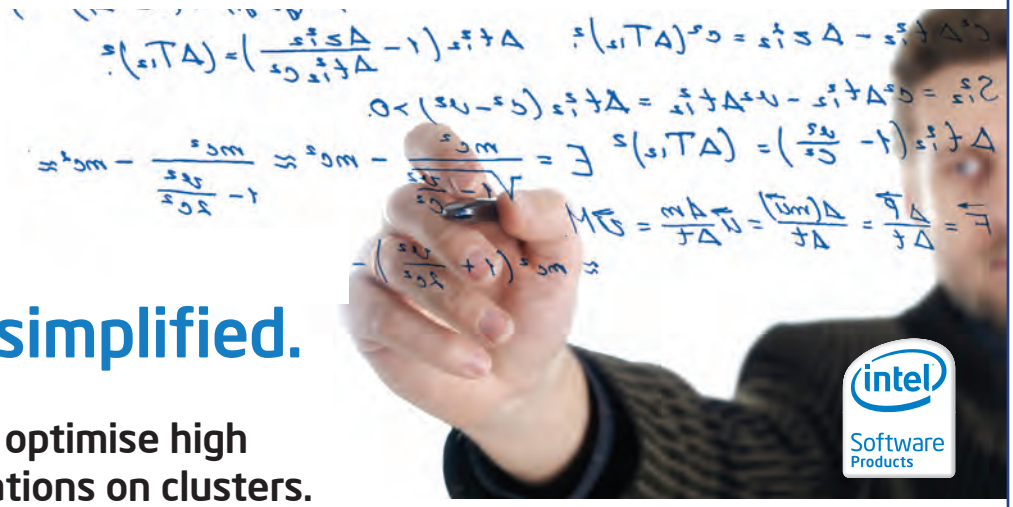
[Stackoverflow] http://stackoverflow.com/questions/286427/calculating-permutations-in-f

Richard Polton is currently pushing paper at HSBC. He has been around the block (of Canary Wharf) all the while moving further away from the True Path of the Zen Coder and was very pleased to pick up Overload 95 and see the Countdown problem posed without a solution. Moreover, he has been lowering the tone since, well, forever really. He can be contacted at drboots@galactichq.org and looks after a splendid web site at http://www.40wc.org/

I have not proven this works, only shown that it appears to work for the example case

```
open System;;
(* modified from stackoverflow *)
let calcPermutations list =
 let rec permutations list taken = [
 if Set.count taken = List.length list then yield [] else
  for l in list do
   if not (taken.Contains l) then
    for perm in permutations list (Set.add l taken)  do
     yield l::perm ]
 permutations list Set.empty;;
let rec op (x:(string * int) list) (y:(string*int)) =
 match (x,y) with
 | ((hd1,hd2)::tl,(y1,y2)) ->
   [ (String.Format("({0}+{1}={2})",hd1,y2,hd2+y2),hd2+y2);
     (String.Format("({0}-{1}={2})",hd1,y2,hd2-y2),hd2-y2);
     (String.Format("({0}*{1}={2})",hd1,y2,hd2*y2),hd2*y2);
     (if hd2%y2=0 then
       (String.Format("({0}/{1}={2})",hd1,y2,hd2/y2),hd2/y2)
      else
       ("non-integer",0))] @ op tl y
 | ([],y) -> [];;

let countdown a b c d e f =
 calcPermutations [a;b;c;d;e;f] |> List.map
 (fun x ->
   match x with
    | [i1;i2;i3;i4;i5;i6] ->
     op (
      op (
       op (
        op (
         op
          [(String.Format("{0}",i1),i1)]
          (String.Format("{0}",i2),i2))
         (String.Format("{0}",i3),i3))
        (String.Format("{0}",i4),i4))
       (String.Format("{0}",i5),i5))
      (String.Format("{0}",i6),i6)
    | _ -> [("end",0)]);;
(* dummy pattern to quiet the compiler *)

let desiredResult = 195;;
let range=5;;

let doIt i1 i2 i3 i4 i5 i6 =
 List.map (List.filter (fun ((x:string),(y:int)) ->
 not( x.Contains("non-integer"))
  && abs (y-desiredResult)<range))
  (countdown i1 i2 i3 i4 i5 i6);;

doIt 1 2 3 4 5 75;;
```

Listing 1



Complexity simplified.

Create, analyse and optimise high performance applications on clusters.

Compile and tune for HPC processing. Create the fastest software possible and utilise the latest technologies within Intel architecture.

Intel® Compiler Suite 11 Professional Edition for Windows* and for Linux* bundles the Intel C++ and Visual Fortran compilers with Intel Math Kernel Library (Intel MKL), Intel Integrated Performance Primitives (Intel IPP), and Intel Threading Building Blocks (Intel TBB).

Intel® Cluster Toolkit Compiler Edition 3.2 provides an extensive software package containing Intel C++ and Intel Fortran Compilers for all Intel architectures, PLUS all the Intel Cluster Tools that help you develop, analyse and optimise performance of parallel applications on Linux* or Windows* Compute Cluster Server (CCS) clusters.

Intel® Math Kernel Library is a highly optimised numerical processing library for math, scientific, engineering and financial applications.

Intel® MPI Library provides a flexible implementation of MPI for easier message-passing interface development on multiple network architectures.

> Within a decade, a programmer who does not think 'parallel' first will not be a programmer. "
>
> James Reinders
> Chief Software Evangelist of Intel Software Products

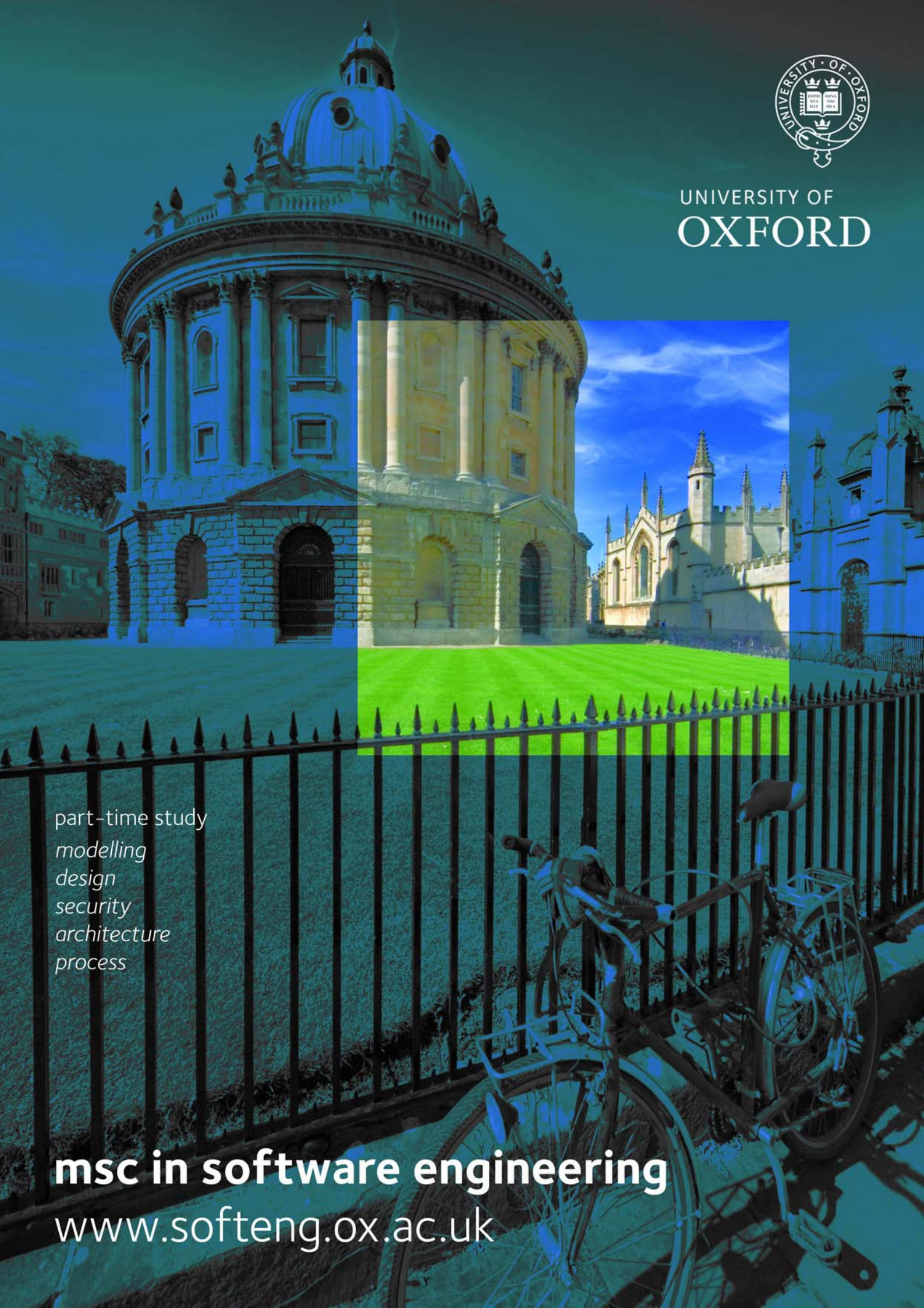
QBS SOFTWARE

TAKE THE NEXT STEP

Visit us at www.qbssoftware.com/intel

Tel: 08456 580 580

intel Software Elite Reseller


UNIVERSITY OF
OXFORD
part-time study
modelling
design
security
architecture
process
msc in software engineering
www.softeng.ox.ac.uk