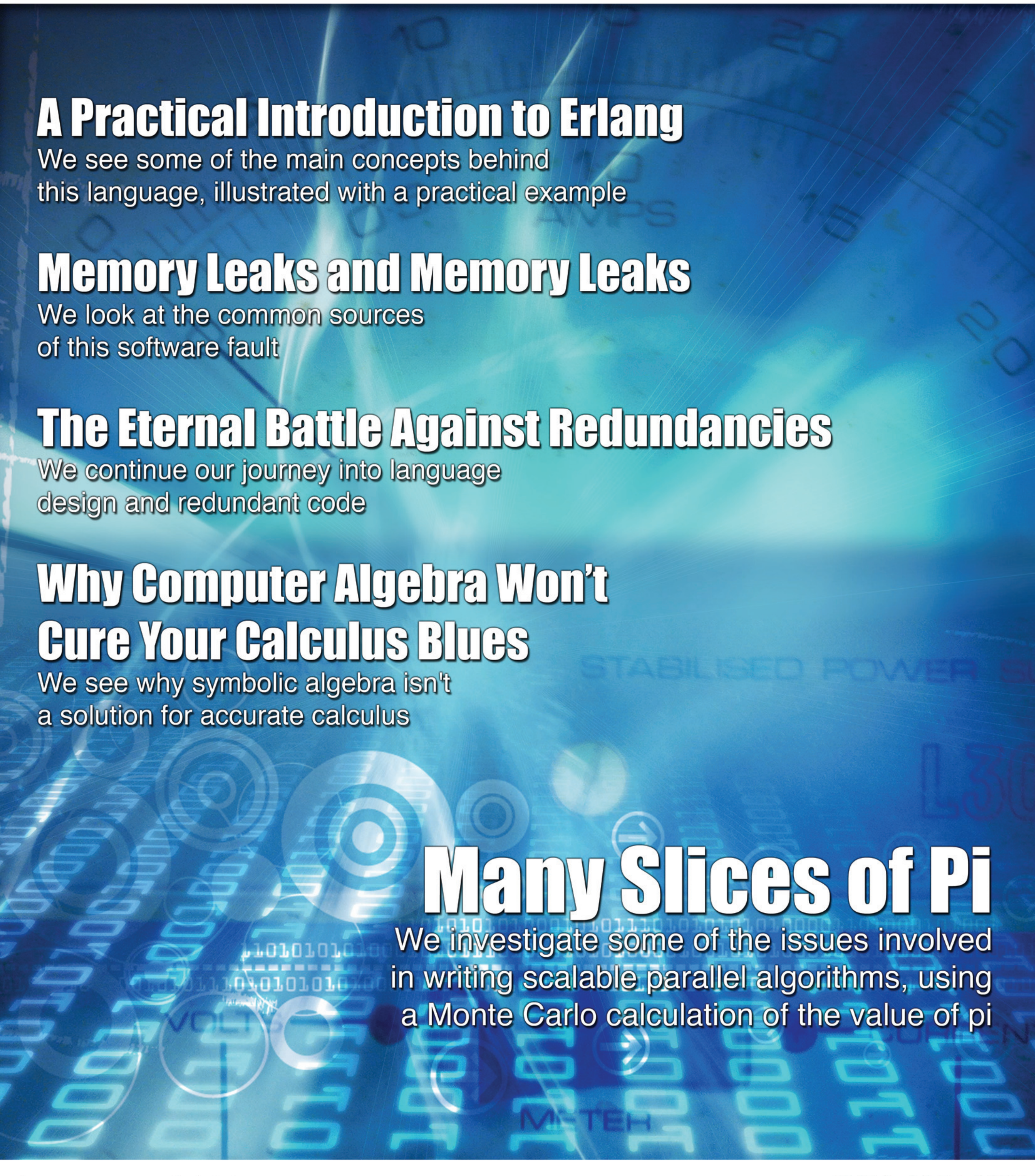
overload 107

FEBRUARY 2012 £3



A Practical Introduction to Erlang
We see some of the main concepts behind this language, illustrated with a practical example

Memory Leaks and Memory Leaks
We look at the common sources of this software fault

The Eternal Battle Against Redundancies
We continue our journey into language design and redundant code

Why Computer Algebra Won't Cure Your Calculus Blues
We see why symbolic algebra isn't a solution for accurate calculus

Many Slices of Pi
We investigate some of the issues involved in writing scalable parallel algorithms, using a Monte Carlo calculation of the value of pi

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 107

February 2012

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Richard Blundell
richard.blundell@gmail.com

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 108 should be submitted by 1st March 2012 and for Overload 109 by 1st May 2012.

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

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.

Many Hands Make Light Work

Some people say the parallel revolution is coming. Ric Parkin argues it's in full swing.

So what do you know about Moore's Law [Moore]? Despite being remarkably famous for a technical 'law', people tend to misquote it or confuse it with related corollaries. The underlying idea is interesting for its simplicity, and the fact it's held true for so long. The original formulation is surprisingly technical [Intel] and deals with the change in the minima of the cost/density curve of packing transistors on a single wafer. Simply, there is a 'sweet-spot' for transistor density cost – making fewer is more reliable but you get less, but adding more can reduce yields. This curve has a minimum where you get the best complexity value. Moore's observation was that this ratio was doubling every year and looked to roughly continue at this rate for at least the immediate future; this was in 1965, so it's gone on for a while! Further refinements from data changed this to a doubling every two years, and a minor refinement by David House, taking into account the improvement in speed due to smaller transistors, tweaked this to 18 months. This is more subtle than the usual formulations: the most common one is that the number of transistors doubles. This isn't a bad approximation though, if you consider how much you get at a constant 'middle' price-point.

Increase in clock speed

Another formulation that has fallen out of favour was that processing speed doubled. This was roughly true for a long time, partly due to the performance increase that House noted, but depended a lot on the increase in clock speed, which juddered to a halt shortly after 2000, as noted by Herb Sutter in 'The Free Lunch is Over' [Sutter1]. It's interesting to note that a major contribution to this was the problem of heat dissipation from the activity of transistors, a problem noted by Moore himself in his paper!

But the halt of the almost-free performance boosts due to clock speed didn't stop the underlying increase in transistor density. You had more to work with, but things no longer got faster as if by magic. Instead these extra transistors were used to implement ingenious shortcuts – things like:

- Pre-fetching instructions that might be needed in the near future.
- Speculative execution where you work out what these instructions would do, without committing the results until you found they'd been run. (This is why having lots of branches can hurt performance – it interferes with this optimisation. Hence tricks such as loop unrolling.)
- Data caching in on-chip memory to get around the comparatively slow communication between the chip and the main memory, by keeping the data you're working on nearby on fast memory on the chip itself.
- Memory write reordering, where it's faster to write to memory in a single pass so the hardware changes the order to be more efficient.
- Parallel execution of independent instructions.

But a we've run out of a lot of those clever tricks. Instead those extra transistors are being used to create extra cores that can truly run multiple threads of execution simultaneously rather than switching from one thread to another, as happens when multi-threaded programs get run on single cores. However, this had some odd effects on some already compiled programs that used multiple threads: they slowed down! Understanding why this could happen is instructive. Remember how on-chip caches were used to avoid the slow round-trip to main memory and back? Well a similar thing happens between cores: if you share data between two cores that are running communicating threads, then keeping their data in sync with each other takes time. Also the locking of that data can cause a core to stall completely while someone else is modifying it. This is not an efficient use of the hardware. So one thing to learn is to be very careful about how many threads you spawn, and how much they need to communicate or share data. It's all too easy to assume that you're the only thing running, spawn more threads than cores and end up slowing everything down due to contention. In fact one efficient arrangement for many standard applications is to have your program single threaded! That way you play 'nicely' with other applications that can be competing for resources. A slight variant on this would be to have all the user interfaces running on one core, and the real 'work' algorithms running on a second core, only occasionally interacting with the UI. Indeed, this is how some operating systems work, in order for their user interfaces to be slick and responsive. Another hard learnt lesson is to avoid sharing data – it's often counter-intuitively faster to make a deep copy to pass to a worker thread than share data. (Functional languages have a big advantage here as they have a much better idea that things won't change and can do an efficient copy if needed.)

Specialised chips

Another long-established trend is to use extra specialised chips to support some computing-intensive operations and relieve the more general purpose CPUs. My first experience of such chips were in early PCs where you could have a separate floating point unit, such as the Intel 80287 [80287], to boost performance of maths-intensive programs. Otherwise floating point had to be done by the main CPU using slow emulation libraries. (I remember stepping though the Borland Pascal libraries and spotting that the start of their maths library checked for the presence of a FPU, and then changed all the functions' code to either use it and return immediately, or implement the software version. This neatly avoided a

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.

check every time or a level of indirection, but such self-modifying code is frowned upon nowadays.)

Another major use for separate specialised chips is for boosting graphics performance. Large displays need a lot of memory, and modern effects and applications need to modify it quickly in often simple but repetitive ways. Taking such a burden off the CPU and giving it to a dedicated chip with a large amount of on-chip memory, and many small cores that can work on different areas of the screen, can make a huge difference. In the last few years such chips have become enormously powerful, and tweaked so that their often idle computing power can be harnessed for more general purpose processing, helped by specially written libraries. The performance increase can be remarkable for suitable applications, such as databases [Alenka], or even parts of an OS [KGPU].

As with many such special purpose chips, increases in transistor density often means that such facilities get built into the main CPU chip, such as Intel's multimedia instruction set, or the Cell processor as used in the PlayStation 3 [Cell]. However, this can only go so far. While Moore's law still holds, the effort needed to make every smaller transistors in the face of thermal and quantum effects is rising and ultimately there are hard physical limits – although you can get pretty small [Physorg].

Remote processing

So how do you keep increasing computing power? Well, we've already seen one way – add more transistors. But not necessarily on the same chip – GPUs and FPUs were originally on separate chips, and we can continue in this tradition by adding multiple chips on each board, or add more boards to the computer, or even add more computers. Herb Sutter has again mapped out this trend in an excellent overview [Sutter2]. Put simply, we no longer have the old traditional single CPU and single thread at our disposal – we now have multiple-threads on multi-core chips, in computers with several chips of various types, connected to a network that can contain millions of other computers, many of which could be ours to use, or possibly rented from the cloud on a ongoing or ad hoc basis. Example of what this can do is shown by Apple's Siri voice recognition services and Amazon's Silk browser – rather than relying on the rather puny computing resources of a mobile phone, these applications get serves in the cloud to do the hard work. In Siri's case it sends the voice stream over the network where a server will do the recognition and work out what to do. In Silk, the web request is actually done by their server, and the final page image is rendered and returned for display.

Looking to the future

So future applications will often be highly complex clusters of algorithms, split between a relatively low powered display and entry device, and an amorphous cloud of computing nodes, which can come and go as needed. Of course there is a problem with reliablity – if the network connection goes, so does your computing power!

Does this sound complex? It sure is, as anyone who's tried designing algorithms for multithreading. This is even more complex – the problem is we'll have a huge amount of unpredictable hardware available and our software will have to adapt to what it finds. Ultimately this has to mean that we no longer deal with low level threading, locking etc, and instead build upon a higher level platform which works out how best to distribute the computing needs. Some languages and platforms such as Erlang have already made some progress in this direction. Other languages such as C++ have only just reached the low level thread and hardware stage – the time is ripe for higher abstractions and language features to give us the vocabulary to describe out algorithms correctly for the future. ■

References

[80287] http://en.wikipedia.org/wiki/80287
[Alenka] http://sourceforge.net/projects/alenka
[Cell] http://en.wikipedia.org/wiki/Cell_(microprocessor)
[Moore] http://en.wikipedia.org/wiki/Moore's_law
[Intel] ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf
[KGPU] http://code.google.com/p/kgpu/
[Physorg] http://www.physorg.com/news193896845.html
[Sutter1] http://www.gotw.ca/publications/concurrency-ddj.htm
[Sutter2] http://herbsutter.com/2011/12/29/welcome-to-the-jungle/

Memory Leaks and Memory Leaks

Correct use of memory is a major occupation of software development. Sergey Ignatchenko considers what we mean by 'correct'.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with opinions of the translator or *Overload* editors; please also keep in mind that translation difficulties from Lapine (like those described in [LoganBerry2004]) might have prevented from providing an exact translation. In addition, both translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Memory leaks are one big source of problems which have plagued both developers and users for generations. Still, the term itself is not as obvious as it might seem, so we'll start from the very beginning: how should a *memory leak* be defined?

Definition 1: the user's perspective

I shall not today attempt further to define the kinds of material I understand to be embraced . . . But I know it when I see it . . .
Justice Potter Stewart on the definition of obscenity

The first point of view we'd like to mention is the one of the user. It is not that easy to define, but we'll try nevertheless. Wearing the user's hat, I would start with saying that a 'memory leak is any memory usage which I, as a user, am not interested in'. This one is probably a bit too broad (in particular, it will include caches which are never in use), so I (still wearing the user's hat) will settle for a less all-inclusive *definition 1*:

> A memory leak is any memory which cannot possibly be used for any meaningful purpose.

Definition 2: the developer's perspective

In developer (and computer science) circles, definitions similar to *definition 2* are quite popular:

> A memory leak is any memory which is not reachable.

Here 'reachable' is a recursive definition, and 'reachable memory' is memory which has a reachable pointer to it – or stack, and 'reachable pointer' is a pointer which resides within reachable memory.

This definition is much more formal than our *definition 1* (and therefore it is much easier to write a program to detect it), but is it a strict equivalent of *definition 1*? Apparently, it is not: let's consider the Java program (Program 1) in Listing 1.

According to *definition 2*, there is no possible memory leak in Java (the garbage collector takes care of unreachable objects). Still, according to *definition 1* there is a memory leak. It illustrates that *definition 1* and *definition 2* are not strictly equivalent: at the very least, *definition 1* has elements which are not members of *definition 2* (see Figure 1).

```
Vector bufs = new Vector();
while( true )
{
  String in = System.console.readLine( "..." );
  if( in == "*" )
    break;
  byte buf[] = new byte[ 1000000 ];
  bufs.add( buf );
  // do something with buf
}
//bufs is not used after this point
```

Listing 1

It should be mentioned that, obviously, Program 1 shows just one trivial example, and much more sophisticated examples of such behaviour are possible (for example, code may allocate huge objects in response to some events, and forget to clean them up until some later event where these objects will be simply discarded without ever reading them).

Definition 3: the debugger's perspective

Going even further into formalism, let's consider a very popular way of memory leak detection deployed by many programs (from Visual Studio to Valgrind). These programs tend to keep track of all allocations and deallocations (either within the heap itself, or otherwise) and report whatever has not been deallocated at the program exit as a memory leak. This leads us to definition 3:

> A memory leak is memory which has not been deallocated at the program exit.

It is fairly obvious that according to this definition, Program 1 doesn't suffer from memory leaks, so *definition 3* is not equivalent to *definition 1*, and some of situations described as leaks by *definition 1*, are not leaks by *definition 3*. But can we say that all situations described as leaks by *definition 3*, are leaks by *definition 1*? Apparently, we cannot. Let's consider another program (Program 2) which allocates a buffer of 4K at the very beginning, uses it through the life cycle of the program and doesn't deallocate it ever, relying on the operating system to clean up after the program terminates. Is it a memory leak? According to *definition 1* (and assuming that our Program 2 runs under an OS which performs cleanup

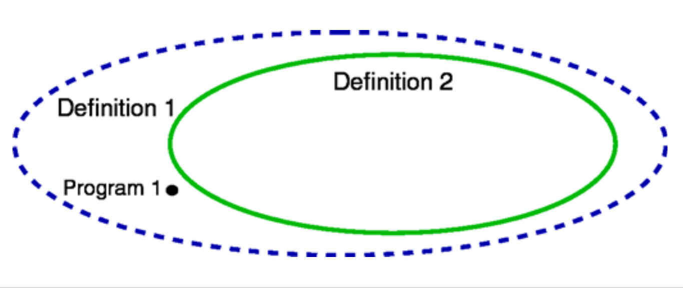


Figure 1

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

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

why should I spend my CPU cycles on performing unnecessary clean up work?

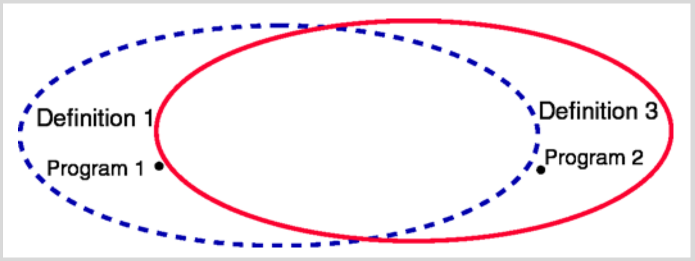


Figure 2

correctly) it is not; according to *definition 3*, it is. It leads us to the relationship between *definition 1* and *definition 3* shown in Figure 2.

Which definition is better?

Up to this point we haven't asked ourselves which of the definitions is better and under which circumstances. We were merely trying to demonstrate that there are substantial differences between them. Now it is time to make a choice.

Remembering the teachings from an earlier article [Bunny2011], we argue that the only correct definition is the one which comes from the User; this is not to diminish the value of tools like Valgrind, but to help to deal with situations when there is a disagreement over whether a certain behaviour is a leak or not.

Some time ago I was in a rather heated debate about a certain program. That program did indeed allocate about 4K of memory at the start (for a good cause, there was no argument about it) and did not bother to deallocate it at all. Obviously Visual Studio had reported it as a leak, and obviously there were pious developers who took Visual Studio's leak reporting as gospel and argued that it was a bug which must be fixed. However as a fix would be non-trivial (in a multithreaded environment, dealing with deallocating globals is not trivial at all) it would likely cause real problems for end-users, and so I was arguing against the fix. Now, the answer to this dilemma is indeed rather obvious: in case of any disagreements between the various definitions of memory leaks it is *definition 1*, and not any other definition, which should be used to determine if program behaviour qualifies as a leak.

Going a bit further we can ask ourselves – what exactly is the purpose of all those deallocations at the end of the program? Why not simply call `ExitProcess()` or `exit()` after all necessary disk work has been completed and all handles closed? Sure, it is sometimes better to simply call all destructors for the sake of simplicity (and therefore, reliability), but on the other hand, if I'm a user why should I spend my CPU cycles on performing unnecessary clean up work? To make things worse, if the program uses lots of memory then a lot of it is likely to have been swapped out to virtual memory on the disk. So to perform the unnecessary deallocations, it will need to be swapped into main memory causing significant inconveniences to the user (if you have ever wondered why closing a web browser takes minutes – this is your culprit). To summarize our feelings on this issue of deallocation at the end of the program – we do not argue that `ExitProcess()` or equivalent is the only way to handle the issue, but we argue that it is one of the possible ways which at least in some cases has a certain value (especially if full-scale deallocation is still performed during at least some test runs to detect real memory leaks). One reasonable solution, from our point of view, would be to try to have all destructors and deallocations in place, and to run all the tools in debug mode, while resorting to `ExitProcess()` or equivalent in release; while there is a drawback that release mode becomes not quite equivalent to debug mode, in many cases it can still be tested properly (especially if QA tests the release version).

Formalism results in approximation

The whole story of multiple definitions of memory leaks is quite interesting if it's viewed from a slightly different (and less practical) angle. We can consider definition 2 as a formal approximation of the much less formal definition 1; as we've seen above this approximation is apparently not 100% precise.

Further, we can consider definition 3 as a further, even more formal, approximation of definition 2, and once again this is still an approximation, and again it is not 100% precise. This leads us to an interesting question: is it necessary that adding more formalism leads to a loss of original intention? ■

References

[Bunny2011] 'The Guy We're All Working For', Sergey Ignatchenko, *Overload* #103

[LoganBerry2004] David 'Loganberry', 'Frithaes! – an Introduction to Colloquial Lapine!', http://bitsnbobstones.watershipdown.org/lapine/overview.html

Many Slices of π

Many numberic estimation techniques are easily parallelisable. Steve Love employs multi-threading, message passing, and more in search of π.

You have to tell a complete story and deliver a complete message in a very encapsulated form. It disciplines you to cut away extraneous information.

~ Dick Wolf, on Advertising

The Monte-Carlo simulation is a common occurrence in computing, used as a way of 'guess-timating' some outcome through repeated sampling. Very often, simulations are processor and memory intensive, performing millions of calculations. The idea is simple: do the same (possibly small) calculation lots of times, usually with random inputs (called sampling), and aggregate all the results in some way.

A larger simulation (more calculations) generally has more accurate results, and so being able to scale in terms of time and space is of great importance. As the uses of calculation services become more sophisticated and demand increasingly precise and timely results, the problem remains the same: how to provide more accuracy in less time. Which means making better use of available resources.

To illustrate the concept, this article takes the example of estimating using a simulation. Although it's a simple enough calculation, it demonstrates some techniques to make a calculation 'engine' scale well with available resources, and examines some of the trade-offs which are inevitable.

Give me π

Estimating π using a simulation is a straightforward enough calculation. The principle is as follows:

> The area of a circle inscribed within a square has a ratio of π/4 to the area of the square. If a number of items of equal size are dropped randomly within the square, then the ratio of the number of items within the circle to the total number of items is (approximately) π/4.

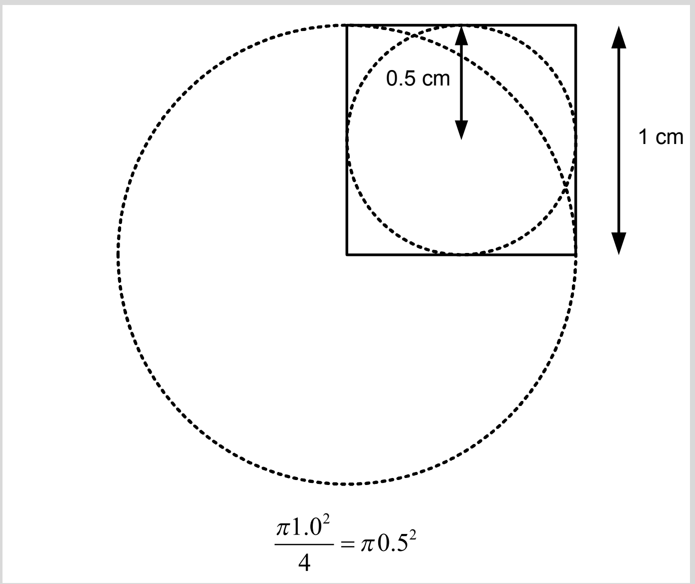
In a computer program, dropping items is a matter of getting a pair of random numbers to represent x and y coordinates within the square. For this example, pseudo-random numbers uniformly distributed over a given range is random enough. Assuming a unit-square, then two random numbers between 0.0 and 1.0 provide the necessary co-ordinates. This is the sampling of the data.

Determining if that co-ordinate is also inside the circle requires reference to Pythagoras:

> The square of the length of the hypotenuse of a right-angled triangle is equal to the sum of the squares of the lengths of the other two sides.

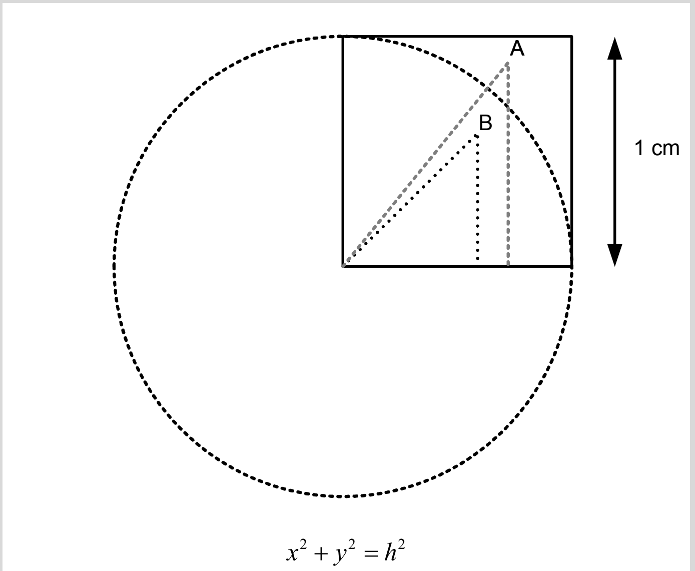
Or, perhaps more memorably, $x^2+y^2 = h^2$. Note that the area of quarter of a circle with radius 1.0 is the same as the area of a circle with radius 0.5 (see figure 1). It is a simple matter to calculate the length of the hypotenuse of the triangle formed by the point at the x,y co-ordinates, the projection of that point on to the x-axis and the origin of the square (the centre of the circle). See figure 2. If the length of the hypotenuse is greater than 1.0 (the radius of the circle), then the location is not within the circle.

Using many such random co-ordinates (samples), the ratio of those within the circle to the total number of samples should be approximately π/4.



$$\frac{\pi 1.0^2}{4} = \pi 0.5^2$$

Figure 1



$$x^2 + y^2 = h^2$$

Figure 2

Steve Love is an independent developer constantly searching for new ways to be more productive without endangering his inherent laziness. He can be contacted at steve@arventech.com

As with most Monte Carlo scenario calculations, we want to aggregate the results of many simulations by taking the mean average

```
public static int Simulate( int count )
{
  var hits = 0;
  var rnd =
     new Random( ( int )DateTime.Now.Ticks );
  foreach( var i in Enumerable.Range( 0, count ) )
  {
    var xSq = Math.Pow( rnd.NextDouble(), 2 );
    var ySq = Math.Pow( rnd.NextDouble(), 2 );
    if( Math.Sqrt( xSq + ySq ) <= 1.0 )
      ++hits;
  }
  return hits;
}
```

Listing 1

Listing 1 shows C# code to sample the co-ordinates. Listing 2 shows the code to calculate the ratio of hits within the circle and multiply up by a factor of 4 to estimate. [MonteCarlo]

Running the simulation code with a sufficiently large sample size should produce a reasonable estimate of π. The larger the sample size, the more accurate the result is likely to be.

Of course, the **Simulate** function could assume a simulation size of 1, and the calling code could just invoke it many times, but this turns out to be – unsurprisingly – colossally inefficient.

This isn't really a Monte-Carlo simulation as such; the next stage is to perform many such simulations and aggregate the results.

Average π

As with most Monte-Carlo scenario calculations, we want to aggregate the results of many simulations by taking the mean average. Naively, the method for this is straightforward. If we have n results, then the mean is calculated by summing all the results and dividing by *n*.

As noted previously, however, the greatest benefit of performing a Monte-Carlo simulation is obtained by making *n* as large as possible, with the result that taking the mean that way can cause issues with such things as overflow, not to mention the overhead of having to store all of the results before the mean can be calculated.

Listing 3 shows how a running average can be taken at each step, effectively eliminating the problem.

```
public static
   double EstimatePi( int hits, int count )
{
  return 4.0 * hits / count;
}
```

Listing 2

```
public static double RunningAverage( int count,
   double last, double next )
{
  return last + ( next - last ) / ( count + 1 );
}
```

Listing 3

It's now practical to put these bits of code together and perform a Monte-Carlo simulation. Listing 4 demonstrates a single-threaded version, along with the results of four runs.

In this example – and all the following examples – *n* is quite small, just 100, whilst the sample size is relatively large. The purpose of that will become apparent later, but for now it is merely enough to note that it's sufficient to estimate π fairly well.

Shared π

In order to make better use of any multi-core or multi-processor resources, a multi-threaded version seems an obvious next step. Listing 5 shows one way that might be done.

The work to be done is split between 4 threads, each doing one quarter of the necessary work, with the running average calculated by each thread.

Since the results are written to a shared resource (the variable **pi**), access to it must be synchronised safely, resulting in the need to lock. Additionally, the code must explicitly wait for all of the launched threads to complete before attempting to read the final value of the pi; not only would an early read be getting an incomplete value, it also introduces a race-condition.

Arguably, the simple **RunningAverage** function might be better encapsulated as a simple class, but this would add an extra point in the code

```
var simsize = 999999;
var count = 100;

var pi = 0.0;
foreach( var i in Enumerable.Range( 0, count ) )
{
  var hits = Common.Simulate( simsize );
  pi = Common.RunningAverage( i, pi,
     Common.EstimatePi( hits, simsize ) );
}
Console.WriteLine( pi );

3.1414193014193
3.1415431015431
3.14159102159102
3.1414877014877
```

Listing 4

The running average calculation depends upon knowing how many results have been seen so far

```
var pi = 0.0;
var locked = new object();

Action action = delegate
  {
    foreach( var i in Enumerable.Range( 0,
       count / 4 ) )
    {
      var hits = Common.Simulate( simsize );
      lock( locked )
        pi = Common.RunningAverage( i, pi,
           Common.EstimatePi( hits,simsize ) );
    }
  };

var tasks = Enumerable.Range( 0, 4 )
  .Select( id => Task.Factory.StartNew(action) )
  .ToArray();
Task.WaitAll( tasks );

Console.WriteLine( pi );
```

Listing 5

```
var nthreads = 4;

Func< double > action = delegate
  {
    var part = 0.0;
    foreach( var i in Enumerable.Range( 0,
             count / nthreads ) )
    {
      var hits = Common.Simulate( simsize );
      part = Common.RunningAverage( i, part,
         Common.EstimatePi(hits, simsize) );
    }
    return part;
  };

var results = Enumerable.Range( 0, nthreads )
  .Select( id => Task< double >
  .Factory.StartNew( action ) )
  .ToArray();

var pi = results.Select(
   t => t.Result ).Average();
Console.WriteLine( pi );
```

Listing 6

where a lock would be required to protect concurrent access to it since it would need to maintain some internal state.

Nevertheless, this code is far from ideal, but it has a more subtle problem.

The running average calculation depends upon knowing how many results have been seen so far. Since each thread is operating on (conceptually) a quarter of the input range, the average will be skewed. In effect, the mean gets calculated for only a quarter of the simulations.

The promise of π

The standard solution to that particular problem is to ensure that the results are aggregated from *all* the threads when they're done. One way to achieve that is to use a Promise.

This is a common mechanism by which a launched thread *promises* to provide a result, and calling code that attempts to access that result will block until the thread has finished. The version in Listing 6 has each thread calculating the running average for its portion of the input, and then returning that to the calling code, which then takes the average of each thread's result.

The call to `t.Result` calls in the promise from each task; if a task has not completed, this call will block until it's ready. Within the delegate that represents a task, it is the return expression that 'fulfils' the promise for each task.

Apart from the fact that the final calculation is now correct, this version has other benefits:

- there is no longer a shared resource, and thus no need to lock.
- waiting for the tasks' results naturally blocks the caller, removing the need to explicitly join on each task.

It does, however, introduce a second calculation of average.

A better solution would be to have all the results collated at a single point in the code, and have each thread somehow present just the results of the simulation. In this version of the simulator, this could be achieved by each task 'returning' the number of hits, and the final calculation of π using the running average algorithm as previously shown.

However, there is still a drawback to this approach: the collation of the results (getting the final average) can't begin until *all* the tasks have finished.

There is, then, room for improvement.

Queue for π

It would be more ideal if the code to calculate the average could run in parallel to the simulations. The standard solution here is to pass a message from each thread to the collating code, as in listing 7. The running average is specifically designed to take a single item result and calculate a new average for all those seen so far.

In this version, the running average calculations will begin as soon as results start appearing on the queue. Now the calculation of π happens *in parallel* with the simulations themselves.

It doesn't make much difference whether the grid is a **high-availability cluster**, a **tightly coupled internal network**, or **distributed over the Internet**

```
var results = new ConcurrentQueue< int >();
var nthreads = 4;
var pi = 0.0;

Action action = delegate
  {
    foreach( var i in Enumerable.Range( 0,
             count / nthreads ) )
    {
      var hits = Common.Simulate( simsize );
      results.Enqueue( hits );
    }
  };

var tasks = Enumerable.Range( 0, nthreads )
  .Select(
   id => Task.Factory.StartNew( action ) )
  .ToArray();

var n = 0;
while( n < count )
{
  int hits;
  if( results.TryDequeue( out hits ) )
    pi = Common.RunningAverage( n++, pi,
       Common.EstimatePi( hits, simsize )
    );
}
Console.WriteLine( pi );
```

Listing 7

The use of a queue also provides a natural synchronisation; the loop which collates the results will not complete until *all* the results are available.

Altogether a much better solution.

Embarrassing π

It's worth noting at this point that the simulations have become almost embarrassingly parallel; each thread is independent of every other, and in particular doesn't depend on the results of any other thread. The only point of shared contact is the queue into which results are placed. The use of a concurrent queue means there is no *explicit* locking of shared resources, but the locks are there nevertheless.

However the code can still benefit from techniques normally used for embarrassingly parallel problems.

Listing 8 makes use of the `Parallel` class available in .Net 4.0. Instead of explicitly launching threads to handle the simulations, the `ForEach` algorithm dispatches *enough* threads to satisfy the given range. For the first time, there is nothing in the code to specify how many threads will run. The `Parallel` class ensures that the best use of available resources is made to execute the code.

```
var results = new ConcurrentQueue< int >();
var pi = 0.0;

Action< int > action = delegate( int i )
  {
    var hits = Common.Simulate( simsize );
    results.Enqueue( hits );
  };
Parallel.ForEach(
   Enumerable.Range( 0, count ),
   action );
var n = 0;
while( n < count )
{
  int hits;
  if( results.TryDequeue( out hits ) )
    pi = Common.RunningAverage( n++, pi,
       Common.EstimatePi( hits, simsize )
     );
}
Console.WriteLine( pi );
```

Listing 8

Distribute the π

To this point, all of the code to perform the simulations has been *in-process*, making use of multi-threading capabilities and supporting features to make the optimum use of a multi-CPU or multi-core environment.

The problem is that many applications for which Monte-Carlo simulation is appropriate are not as simple as estimating π, and are likely to be much more demanding of the available hardware. The π estimation simulation is, in fact, CPU intensive, but more sophisticated problems may also be memory-bound.

One solution to running simulations where there is insufficient memory to perform many calculations (or possibly more likely, insufficient address space) is to add more memory and address space, by for example upgrading to a 64 bit machine and operating system. This isn't scalable, however, and merely pushes the problem back until yet more memory is required.

A more scalable and general solution is to use more than one machine: a grid.

It doesn't make much difference whether the grid is a high-availability cluster, a tightly coupled internal network, or distributed over the Internet, the major difficulty is distributing, running and communicating with the code. To handle this, some kind of middleware is appropriate.

The middleware used here is 0MQ [0MQ], an open-source messaging library with bindings for many languages, including C#.

```
static void Main()
{
  var simsize = 999999;
  var count = 100;

  using( var context = new Context( 1 ) )
  using( var results =
     context.Socket( SocketType.PULL ) )
  {
    results.Bind( "tcp://*:55566" );
    var n = 0;
    var pi = 0.0;
    while( n < count )
    {
      var recv = results.Recv();
      var hitsString =
         Encoding.UTF8.GetString(recv);
      var hits = int.Parse(hitsString);
      pi = Common.RunningAverage( n++, pi,
         Common.EstimatePi( hits, simsize )
         );
    }
    Console.WriteLine( pi );
  }
}
```

Listing 9

No π?

Listing 9 shows an application that handles collating the results from a simulation. It's notable because the simulation code is nowhere to be seen, and isn't even apparently invoked. Instead, messages are consumed from a message queue, and plugged into the now-familiar running average calculation, until the requisite number of results has been received.

Listing 10 shows the code for the simulations 'engine'. Once again, it makes use of the `Parallel.ForEach` algorithm to internally launch several threads, but instead of attempting to service a number of calculations, it's limited to a number of threads.

The message service connects to an endpoint defined by the collating application (If this sounds counter-intuitive, see sidebar), and 'publishes' its results to that connection.

As with the in-process queuing solution shown in listing 7, there is a natural synchronisation in the collating application (listing 9) in that reading from the queue blocks until there are messages available. Also similar between the two methods is that the collating can begin as soon as there are results available, and so operates in parallel with the simulation service.

Key to the benefits here, though, is that the code in listing 9 is a separate process to listing 10 – each has its own `Main()`. It might not be obvious, but the implication of that is that these two separate programs can be run on different machines – provides that those machines can communicate over a network using the specified port number.[1] Modifying the hostname to be something other than 'localhost' in listing 10 would enable this.[2]

The benefit of each simulation having a large sample size, and using a (relatively) small number of simulations (alluded to in 'Average π') should now be evident. When operating in a distributed environment, it's important for performance that the cost of the calculation is not swamped by the cost of 10 the distribution, i.e. the relative expense of running the simulation is worth the effort to communicate over a network.

1. Any free port number can be used
2. The `*` used in the binding code of listing 9 indicates that the connection can use any available network interface. The behaviour of other settings is dependent on the environment, but follows the local socket library conventions.

Endpoints

Just as with raw sockets, it is permitted to connect multiple servers to an endpoint, but it is not possible to bind the same endpoint multiple times. For the purposes of 0MQ, an endpoint is a host and a port number.

The client-server relationship is normally defined by servers having wellknown endpoints, to which many clients can connect. However in the distributed grid-engine world it is often the case that there is a *single* client communicating with several (or even many!) servers.

Sometimes it is desirable to have multiple clients *and* multiple servers. The standard solution to the problems that this poses is to introduce a broker which acts as a static endpoint (server) to both clients and services. 0MQ allows brokers to be very easily constructed, but that's beyond the scope of this article.

Talkin' π

Just because the code has now been changed to *allow* it to be executed in a distributed environment, that doesn't mean it necessarily *must* be.

Listing 11 shows an in-process simulation 'service' running in parallel to the main collating code in the same way as listings 9 and 10.

The difference here is that the Simulator is now being run directly as a thread, and launched in the same way as shown in previous sections (notably in the section called 'Queue for π') which uses a concurrent queue).

Although it's not necessary to do so, the addressing method for the 0MQ sockets has changed; instead of using a 'tcp' protocol definition with a port number (and 'localhost' in the case of the connecting code) the binding and connecting addresses are now symmetric, using the 'inproc' protocol and a symbolic name for the connection.

There are pros and cons to using this approach; a benefit of the 0MQ connection model that has not been mentioned so far is that a 'tcp' socket can be connected that has not yet been bound (*not* possible with raw sockets). The implication of that is important in a distributed environment: the 'engines' can all be running before the collating code (which binds the

```
static void Simulator( Context ctx, int count,
   int simsize )
{
  using( var results =
     ctx.Socket( SocketType.PUSH ) )
  {
    results.Connect( "tcp://localhost:55566" );
    foreach( var i in Enumerable.Range(0, count) )
    {
      var hits = Common.Simulate( simsize );
      results.Send( Encoding.UTF8.GetBytes
         ( hits.ToString() ) );
    }
  }
}

static void Main()
{
  var simsize = 999999;
  var count = 100;
  var nthreads = 4;

  using( var context = new Context( 1 ) )
  {
    Parallel.ForEach(
       Enumerable.Range( 0,nthreads ),
       i => Simulator( context,
          count / nthreads, simsize ) );
  }
}
```

Listing 10

```
static void Simulator( Context ctx, int count,
                       int simsize )
{
  using( var results =
     ctx.Socket( SocketType.PUSH ) )
  {
    results.Connect( "inproc://results" );
    foreach( var i in Enumerable.Range(0, count) )
    {
      var hits = Common.Simulate( simsize );
      results.Send( Encoding.UTF8.GetBytes
         ( hits.ToString() ) );
    }
  }
}

static void Main()
{
  var simsize = 999999;
  var count = 100;
  var nthreads = 4;

  using( var context = new Context( 1 ) )
  using( var results = context.Socket
    ( SocketType.PULL ) )
  {
    results.Bind( "inproc://results" );

    var tasks = Enumerable.Range( 0, nthreads )
      .Select
       ( id => Task.Factory.StartNew(
          () => Simulator( context,
                 count / nthreads,
                 simsize ) ) )
      .ToArray();

    var n = 0;
    var pi = 0.0;
    while( n < count )
    {
      var recv = results.Recv();
      var hitsString =
         Encoding.UTF8.GetString(recv);
      var hits = int.Parse(hitsString);
      pi = Common.RunningAverage( n++, pi,
         Common.EstimatePi( hits, simsize )
         );
    }
    Console.WriteLine( pi );
  }
}
```

Listing 11

endpoint) has been launched. Using ‘inproc’, the endpoint *must* be bound before downstream connections can connect, or a runtime error occurs.

It is perfectly valid to use the ‘tcp’ protocol even in-process, and the performance degradation is minimal. Mileage, as ever, may vary between convenience and raw speed.

A minor problem

With one exception, the multi-threaded and distributed examples so far exhibit a common problem that has been consciously ignored. It is most noticeable in the original multi-threaded queue version in ‘Queue for π’.

Listing 12 shows the offending code from Listing 7. The problem really is minor, but significant.

If the number of simulations to perform is not an exact multiple of the number of threads used to service the requests, some further processing is required to mop up any remainder. This isn’t a difficult problem to solve but it suffers from two drawbacks:

1. It is very easy to get *wrong*
2. It duplicates code.

The exception so far has been Listing 8. There was no reference there to a particular number of threads; the **`Parallel.ForEach`** algorithm handled the mechanics of launching *sufficient* threads to service each element of the range it was given.

This concept doesn’t translate well into a distributed grid environment. Dynamically allocating engines to handle workload is difficult and error-prone. The out-of-process simulation shown in Listing 10 is internally multi-threaded. It doesn’t attempt to dynamically manage the number of threads, but launching multiple instances of that process would affect the characteristics of what are (in that example) hard-coded values for the number of simulations to run versus the number of threads within each process.

Of course, this problem is exacerbated by the fact that the reading client depends on receiving a pre-determined (and common) number of results. It’s probably desirable for the engines to have no knowledge of how many results they must produce.

Slimmer π

The real answer to this problem is to effectively return to single threaded processing.

Really.

Single threaded code is easier to write and easier to understand, and is easy to *prove* correct, so it has lots of built-in benefits. It does, of course, appear – on the face of it – to be at odds with making the best use of multi-CPU, multi-core and multi-engine processing.

In a distributed environment, however, even though each process is single threaded, there can be *many processes*. The use of a shared message queue that isolates each process from the code that handles the results, as well as other processes, does require a slightly different approach to designing the code.

One π at a time

The code in listing 13 shows a simulation engine that runs a single-threaded service. The program itself dispatches more than one thread simply so that the engine can be stopped ‘nicely’ without just killing it.

The beating heart of this code is the one-line while loop near the end of the **`Simulator`** method. It’s the lambda function attached to the **`workEvent.PollInHandler`** that does the work, however.

In truth, it is not so different to other versions of this code shown previously. Its main distinction is the idea of a *message* indicating a work item. In this case, a work item simply tells the engine how large a sample size to use for a single calculation. The engine does that calculation and then sends the result on a a different channel. Then, the code waits for the next work item, and will (hopefully) continue to run indefinitely until manual intervention stops it.

The vast majority of the remainder of the code is to setup the network connections, manage object lifetimes and handle graceful termination of the process.

```
Action action = delegate
  {
    foreach( var i in Enumerable.Range( 0,
             count / nthreads ) )
    {
      var hits = Common.Simulate( simsize );
      results.Enqueue( hits );
    }
  };
```

Listing 12

As many instances of this process can be launched as necessary – on different machines if required, by modifying the 'localhost' address – to make the most of the available resources.

Having an engine that listens for work requests requires the existence of code to *provide* those requests. Listing 14 shows this in action.

Once again a separate thread is used to send work requests so that collating the results in **`Main()`** can start as soon as results are available.

The π message

Anyone familiar with message-passing paradigms such as Actor model or CSP [Wikipedia] will recognise (broadly speaking) the code in the previous section. The 0MQ sockets are channels, with two incoming channels (one for work items, one for a 'stop' message) and one outgoing channel. The Simulator function is a 'process' or 'actor' that runs indefinitely.

Crucially the main idea of passing messages in systems using (for example) CSP is that that is the *only* way that processes communicate with each other.

The model here is more like Actor than CSP because the channels are buffered and asynchronous, whereas CSP channels involve a 'rendezvous' between sender and recipient. Also, the process has no identity. However, in common with CSP, messages are sent to channels with names; in this example the name is the address used to connect or bind a socket.

Apart from receiving the **`Context`** (a thread-safe socket factory) in its parameter list, the process interacts with the outside world only through the channels. There is no thread synchronisation and no shared state. The process runs until it's told to stop – by receiving a message on a particular channel.

The difference from other common message-passing schemes in the CSP or Actor Model style is that messages can be passed on those channels between different machines.

The downside to this distribution is the loss of type-safety in messages; the content of each message needs to be agreed between clients and servers – often by convention. Some middleware tools provide a full Remote

```
static void Simulator( Context ctx )
{
  using( var work =
     ctx.Socket( SocketType.REP ) )
  using( var results =
     ctx.Socket( SocketType.PUSH ) )
  using( var done =
     ctx.Socket( SocketType.PAIR ) )
  {
    done.Connect( "inproc://done" );
    results.Connect( "tcp://localhost:55557" );
    work.Connect( "tcp://localhost:55556" );

    var finished = false;
    var killEvent =
       done.CreatePollItem( IOMultiPlex.POLLIN );
    killEvent.PollInHandler += ( sock, ev ) => {
       sock.Recv(); finished = true; };

    var workEvent =
       work.CreatePollItem( IOMultiPlex.POLLIN );
    workEvent.PollInHandler += ( sock, ev ) =>
    {
      var simsize =
         int.Parse( sock.Recv( Encoding.UTF8 ) );
     sock.Send( Encoding.UTF8.GetBytes( "OK" ) );
      var hits = Common.Simulate( simsize );
      results.Send( Encoding.UTF8.GetBytes
         ( hits.ToString() ) );
    };

    var items = new []{ killEvent, workEvent };

    while( ! finished )
      ctx.Poll( items );
  }
}

static void Main()
{
  using( var context = new Context( 1 ) )
  using( var done =
         context.Socket( SocketType.PAIR ) )
  {
    done.Bind( "inproc://done" );

    Task.Factory.StartNew(
      () => Simulator( context ) );

    Console.WriteLine( "Press [Enter] to exit" );
    Console.ReadLine();
    done.Send();
    Console.WriteLine( "Done" );
  }
}
```

Listing 13

```
static void Work( Context ctx, int count,
                  int simsize )
{
  using( var work =
     ctx.Socket( SocketType.REQ ) )
  {
    work.Bind( "tcp://*:55556" );
    foreach( var i in Enumerable.Range
       ( 0, count ) )
    {
      work.Send( simsize.ToString(),
                 Encoding.UTF8 );
      work.Recv();
    }
  }
}

static void Main()
{
  var simsize = 999999;
  var count = 100;

  using( var context = new Context( 1 ) )
  using( var results =
         context.Socket( SocketType.PULL ) )
  {
    Task.Factory.StartNew(
      () => Work( context, count, simsize ) );

    results.Bind( "tcp://*:55557" );
    var n = 0;
    var pi = 0.0;
    while( n < count )
    {
      var hits =
         int.Parse( Encoding.UTF8.GetString(
                    results.Recv() ) );
      pi = Common.RunningAverage( n++, pi,
         Common.EstimatePi( hits, simsize ));
    }
    Console.WriteLine( pi );
  }
}
```

Listing 14

```
def work( ctx ):
  work = ctx.socket( zmq.REQ )
  try:
    work.bind( "tcp://*:55556" )
    for i in range( count ):
      work.send( str( simsize ) )
      work.recv()

  finally:
    work.close()

def recv( ctx ):
  results = ctx.socket( zmq.PULL )
  try:
    results.bind( "tcp://*:55557" )
    pi = 0.0
    nresults = 0
    while nresults < count:
      hits = int( results.recv() )
      pi = runningAverage( nresults, pi,
           estimatePi( hits, simsize ) )
      nresults += 1
    print( pi )

  finally:
    results.close()

if __name__ == ’__main__’:
  ctx = zmq.Context( 1 )
  try:
    threading.Thread( target=work,
                      args=( ctx, ) ).start()
    recv( ctx )

  finally:
    ctx.term()
```

Listing 15

Procedure Call paradigm [RPC] whereby the type of a message is defined precisely in a definition language, and additionally there are libraries available that do not provide a transport, only the marshalling of data in a type-safe way.

Mixed π

Using a message passing middleware that provides bindings to multiple languages means that any supported language can be used as either client or server.

Listing 15 shows a Python version of the ‘client’ code to send work requests and collate results. This code will communicate happily with the calculation engines shown in Listing 14. The definition of `runningAverage` is left as an exercise for the reader.

The last π

The benefits of grid computing and message passing are not limited to Monte-Carlo simulations. Any algorithmic problem that can benefit from using the resources of many machines can enjoy the benefits of being designed for message passing, but further than that, programs which use multiple threads can be improved by passing messages instead of sharing state.

Many of the perceived difficulties of writing and maintaining multi-threaded code arise from the sharing of state: deadlock, unwanted serialisation due to locking, context switching. None of these problems arise when passing messages is the only interaction between threads. *All* state is necessarily local to a thread, because it’s not really a thread – it’s a process.

Many tools are available for lots of languages to provide a message-passing environment for programs to use in an inter-thread capacity, but few provide the same facility to enable not just inter-*process* but inter-*machine* message passing with few (if any) changes to the code.

0MQ provides facilities to do exactly that, but at the cost of pure performance. Ultimately, the only way to determine if performance is sufficient for a particular application is to measure and, if necessary, compare results using different technologies. However, it’s always as well to remember that clean, simple and maintainable code to do a job will pay dividends in any case.

Especially if the very clarity and simplicity provides generality and flexibility, too. ■

Acknowledgements

Many thanks to Frances Buontempo for her patient tutoring in basic numeracy and helpful comments.

References

[MonteCarlo] See http://en.wikipedia.org/wiki/Monte_Carlo_method for more details

[0MQ] See www.zeromq.org

[RPC] ICE (http://www.zeroc.com) and CORBA (http://www.omg.org/spec/CORBA) are well known examples of the Remote Procedure Call paradigm.

[Wikipedia] http://en.wikipedia.org/wiki/Communicating_sequential_processes and http://en.wikipedia.org/wiki/Actor_model have more information about CSP and the Actor Model.

cqf.com

CERTIFICATE IN QUANTITATIVE FINANCE CQF

Expand Your Mind and Career

Designed by quant expert Dr Paul Wilmott, the CQF is a practical six month-part time course that covers every gamut of quantitative finance, including derivatives, development, quantitative trading and risk management.

Find out more at **cqf.com**.

ENGINEERED FOR THE FINANCIAL MARKETS

Why Computer Algebra Won't Cure Your Calculus Blues

We still haven't found how to accurately do calculus. Richard Harris revisits an algebraic technique.

We began the second half of this series with a brief history of the differential calculus and a description of perhaps the most powerful mathematical tool for numerical computing, Taylor's theorem, which states that for a function *f*

$$\begin{aligned}
f(x+\delta) &= f(x) + \delta \times f'(x) + \tfrac{1}{2}\delta^2 \times f''(x) + \ldots \\
&\quad + \tfrac{1}{n!}\delta^n \times f^{(n)}(x) + R_n \\
\min\left(\tfrac{1}{(n+1)!}\delta^{n+1} \times f^{(n+1)}(x+\theta\delta)\right) &\leq R_n \\
&\leq \max\left(\tfrac{1}{(n+1)!}\delta^{n+1} \times f^{(n+1)}(x+\theta\delta)\right) \quad \text{for} \quad 0 \leq \theta \leq 1
\end{aligned}$$

where $f'(x)$ denotes the first derivative of f at x, $f''(x)$ denotes the second and $f(n)(x)$ the n^{th} and where, by convention, the 0^{th} derivative is the function itself.

We then used Taylor's theorem to perform a detailed analysis of finite difference approximations of various orders of derivatives and, in the previous instalment, sought to use polynomial approximations to automate the calculation of their formulae.

We concluded with Ridders' algorithm [Ridders82] which treated the symmetric finite difference as a function of the change in the argument, δ, and used a polynomial approximation of it to estimate the value of the difference with a δ of zero.

You will no doubt recall that this was a significant improvement over every algorithm we had previously examined with an average error roughly just one decimal order of magnitude worse than the theoretical minimum.

Given that this series of articles has not yet concluded, the obvious question is whether or not we can close that gap.

The obvious answer is that we can.

Sort of.

One of the surprising facts about differentiation is that it is almost always possible to find the expression for the derivative of a function if you have the expression for the function itself. This might not seem surprising until you consider the inverse operation; there are countless examples where having the expression for the derivative doesn't mean that we can find the expression for the function.

This is enormously suggestive of a method by which we can further improve our calculation of derivatives; get the computer to generate the correct expression for us.

Computer algebra revisited

We first discussed computer algebra as part of our quest to find an infinite precision numeric type [Harris11]. The idea was to represent an expression as a tree rather than directly compute its value, as shown in figure 1 (an expression tree for the golden ratio).

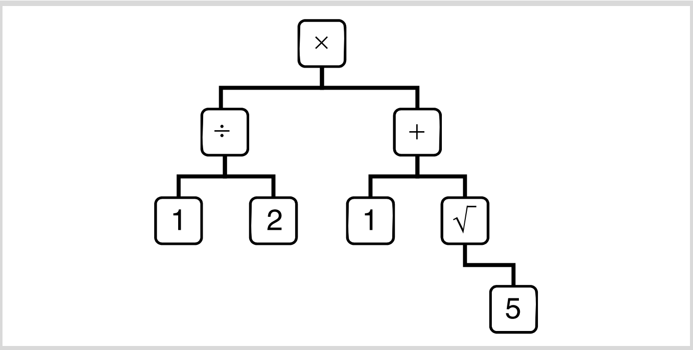


Figure 1

A common implementation of such expression trees uses a pure virtual base class to represent the nodes. Our original attempt is given in listings 1 (the original expression object base class) and 2 (the original expression wrapper class).

You may recall that my proposal to compute the digits of the decimal expansion of the expression one at a time with the **`exact`** member function (with negative indices to the left of the decimal point and positive to the right) as a means of effectively maintaining infinite precision was ultimately doomed to failure because of the impossibility of comparing equal values. There was simply no way, in general, to decide when to give up.

However, the simplicity and near universal applicability of the rules of differentiation gives these expression objects something of a reprieve; they are supremely well suited for automating those rules.

Symbolic differentiation

We shall, of course, need to redesign our expression object classes; they were after all rather useless in their original form.

```
class expression_object
{
public:
  enum{empty=0xE};

  virtual ~expression_object() {};

  virtual double approx() const = 0;
  virtual unsigned char
     exact(const bignum &n) const = 0;
  virtual bignum::sign_type sign() const = 0;
};
```
Listing 1

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.

we shall take it as read that we have defined **a full suite of expression objects** and implemented their value member functions

```
class expression
{
public:
  typedef
     shared_ptr<expression_object> object_type;
  enum{empty=expression_object::empty};

  expression();
  expression(const bignum &x);
  explicit expression(const object_type &x);

  double             approx() const;
  unsigned char      exact(const bignum &n) const;
  bignum::sign_type sign() const;

  object_type  object() const;
  int          compare(const expression &x) const;
  expression & negate();

  expression & operator+=(const expression &x);
  expression & operator-=(const expression &x);
  expression & operator*=(const expression &x);
  expression & operator/=(const expression &x);

private:
  object_type object_;
};
```

Listing 2

We shall therefore do away with the notion of exact evaluation and content ourselves with just floating point and shall further add a virtual member function that returns the expression representing the derivative, as shown in listings 3 (the new expression object base class) and 4 (the new expression wrapper class).

Note that the **`value`** member function replaces the **`approximate`** member function and will have identical implementations in derived classes, as illustrated in listings 5 (the new substraction expression class) and 6 (the **`value`** member function of **`subtraction_expression`**).

```
class expression_object
{
public:
  virtual ~expression_object() {};

  virtual double value() const = 0;
  virtual expression
     derivative(const expression &x) const = 0;
};
```

Listing 3

```
class expression
{
public:
  typedef shared_ptr<expression_object>
object_type;

  expression();
  expression(double x);
  explicit expression(const object_type &x);

  double value() const;
  expression
     derivative(const expression &x) const;

  object_type  object() const;
  expression & negate();

  expression & operator+=(const expression &x);
  expression & operator-=(const expression &x);
  expression & operator*=(const expression &x);
  expression & operator/=(const expression &x);

private:
  object_type object_;
};
```

Listing 4

```
class subtraction_expression :
   public expression_object
{
public:
  explicit subtraction_expression(
     const expression &lhs,
     const expression &rhs);
  virtual ~subtraction_expression();

  virtual double value() const;
  virtual expression
    derivative(const expression &x) const;

  const expression lhs;
  const expression rhs;
};
```

Listing 5

```
double
subtraction_expression::value() const
{
  return lhs.value() - rhs.value();
}
```

Listing 6

we are effectively multiplying one function by the reciprocal of another

```
expression
expression::derivative(const expression &x) const
{
  return object_->derivative(x);
}
```

Listing 7

```
expression
constant_expression::derivative
   (const expression &x) const
{
  return expression(0.0);
}
```

Listing 8

There's little to be gained in repeating this simple change for all types of expression so we shall take it as read that we have defined a full suite of expression objects and implemented their value member functions so that we can concentrate on the implementation of the **`derivative`** member functions.

The **`expression`** wrapper class simply forwards the call to **`derivative`** to the underlying **`expression_object`** as shown in listing 7.

The first of the underlying expression objects we shall consider is the **`constant_expression`**, which should trivially return 0 in all cases, as shown in listing 8.

Next up is the **`variable_expression`** which shall serve as the type we differentiate by. As such it should return 1 if differentiated by itself and 0 otherwise, as illustrated in listing 9.

Addition and subtraction expression object are barely less trivial to differentiate than constants and variables, as illustrated in listing 10.

Now that we've got these trivial cases out of the way we're ready to implement some of the more subtle rules of differentiation. We shall start with the product rule which states that

$$\frac{d}{dx}\left(f\left(x\right)\times g\left(x\right)\right)=f\left(x\right)\times\frac{d}{dx}g\left(x\right)+g\left(x\right)\times\frac{d}{dx}f\left(x\right)$$

```
expression
variable_expression::derivative
   (const expression &x) const
{
  return expression
     (x.object().get()==this ? 1.0 : 0.0);
}
```

Listing 9

```
expression
addition_expression::derivative
   (const expression &x) const
{
  return lhs.derivative(x) + rhs.derivative(x);
}

expression
subtraction_expression::derivative
   (const expression &x) const
{
  return lhs.derivative(x) - rhs.derivative(x);
}
```

Listing 10

This identity is reflected in the implementation of the **`derivative`** member function of the **`multiplication_expression`** class given in listing 11.

Division is slightly more complicated since it relies upon the chain rule which states that

$$\frac{d}{dx}f\left(g\left(x\right)\right)=\frac{d}{dg\left(x\right)}f\left(g\left(x\right)\right)\times\frac{d}{dx}g\left(x\right)$$

In the case of division we are effectively multiplying one function by the reciprocal of another, so we must apply both the chain rule and the product rule.

$$\begin{aligned}\frac{d}{dx}\frac{f\left(x\right)}{g\left(x\right)}&=\frac{d}{dx}\left(f\left(x\right)\times\frac{1}{g\left(x\right)}\right)\\&=f\left(x\right)\times\frac{d}{dx}\frac{1}{g\left(x\right)}+\frac{1}{g\left(x\right)}\times\frac{d}{dx}f\left(x\right)\\&=-\frac{f\left(x\right)}{g\left(x\right)^2}\times\frac{d}{dx}g\left(x\right)+\frac{1}{g\left(x\right)}\times\frac{d}{dx}f\left(x\right)\end{aligned}$$

An implementation is given in listing 12.

The chain rule is absolutely fundamental in working out how to implement the **`derivative`** member function for all of the remaining expression objects.

```
expression
multiplication_expression::derivative
   (const expression &x) const
{
  return lhs * rhs.derivative(x) +
         rhs * lhs.derivative(x);
}
```

Listing 11

we have finally achieved what we set out to do; implement an algorithm that can calculate the derivative of a function to machine precision

```
expression
division_expression::derivative
   (const expression &x) const
{
  const expression &dl = lhs.derivative(x);
  const expression &dr = rhs.derivative(x);

  return -dr*lhs/(rhs*rhs) + dl/rhs;
}
```

Listing 12

As a final example, we shall take a look at raising one expression to the power of another. Now it is not entirely obvious how to do this. Differentiating a variable raised to a constant power should be familiar enough

$$\frac{d}{dx}x^c = c \times x^{c-1}$$

However, differentiating a constant raised to the power of a variable isn't quite so straightforward

$$\frac{d}{dx}c^x = ?$$

The trick is to exponentiate the logarithm of c^x

$$\frac{d}{dx}c^x = \frac{d}{dx}e^{\ln c^x} = \frac{d}{dx}e^{x \ln c}$$

after which the result is trivially

$$\ln c \times e^{x \ln c} = \ln c \times c^x$$

Combining this with the chain rule, the product rule and the fact that the derivative of the exponential is equal to itself yields

$$\begin{aligned}\frac{d}{dx}f(x)^{g(x)} &= \frac{d}{dx}e^{g(x)\times \ln f(x)} \\ &= \frac{d}{dx}\big(g(x)\times \ln f(x)\big)\times e^{g(x)\times \ln f(x)} \\ &= \left(\begin{array}{l}\ln f(x)\times \frac{d}{dx}g(x) + \\ g(x)\times \frac{d}{dx}\ln f(x)\end{array}\right)\times f(x)^{g(x)} \\ &= \left(\begin{array}{l}\ln f(x)\times \frac{d}{dx}g(x) + \\ \frac{g(x)}{f(x)}\times \frac{d}{dx}f(x)\end{array}\right)\times f(x)^{g(x)}\end{aligned}$$

```
expression
power_expression::derivative
   (const expression &x) const
{
  return (log(lhs)*rhs.derivative(x)
     +rhs*lhs.derivative(x)/lhs) *
     expression(expression::object_type(this));
}
```

Listing 13

Listing 13 illustrates an implementation of the derivative of a power expression.

I shall leave the implementation of further expression objects to you and move on to consider the implications of this approach.

Exact differentiation?

On the face of it we have finally achieved what we set out to do; implement an algorithm that can calculate the derivative of a function to machine precision.

In using expression objects to symbolically differentiate expressions we are able to generate an expression that is mathematically identical to the derivative.

All that remains is to evaluate it.

Unfortunately there are a few problems.

Memory use

The first problem is that the expressions representing derivatives can grow in complexity at an alarming rate.

For example, consider the function

$$f(x) = e^{x^2}$$

Differentiating with respect to *x* yields

$$\frac{d}{dx}f(x) = 2x \times e^{x^2}$$

Differentiating again yields

$$\frac{d^2}{dx^2}f(x) = 2e^{x^2} + 4x^2 \times e^{x^2}$$

Once more

$$\frac{d^3}{dx^3}f(x) = 4x \times e^{x^2} + 8x \times e^{x^2} + 8x^3 \times e^{x^2}$$

Clearly the trees representing these expressions will grow quite rapidly unless we algebraically simplify them. For example, the third derivative can be simplified to

There are many mathematical functions for which we have **no simple formulae** and must **instead rely upon approximations**

$$\frac{d^3}{dx^3} f(x) = \left(12x + 8x^3\right) \times e^{x^2}$$

reducing the number of arithmetic operations from 15 to 7.

Unfortunately implementing a computer algebra system that is capable of performing such simplifications is no easy task. Much like chess programs they require large databases of valid transformations of expressions, some heuristic that accurately captures our sense of *simplicity* and expensive brute-force search algorithms to traverse the variety of ways in which those transformations can be applied.

Unless we are willing to expend a very great deal more effort we shall have to accept the fact that symbolic differentiation will be something of a memory hog.

Cancellation error

The next problem also stems from the difficulty in simplifying expressions but is rather more worrying. Consider redundant expression which though complex, when fully simplified, yield trivial expressions.

As an example consider

$$f(x) = \frac{x}{x}$$

Trivially, this has a derivative of 0 for all *x*, but when we apply our expression objects to compute the derivative, they will blindly follow the rules to yield

$$\frac{d}{dx} f(x) = \frac{d}{dx}\left(x \times \frac{1}{x}\right) = \frac{1}{x} - x \times \frac{1}{x^2}$$

Mathematically, this is identically equal to 0 for all *x*, but since we are using floating point some errors will inevitably creep in. We shall consequently be subtracting two nearly equal numbers which is the very incantation we use to summon the dreaded cancellation error.

Figure 2 illustrates the result of this calculation for *x* from 0.01 to 1 in steps of 0.01 and clearly shows the effect of cancellation on the result.

Whilst this is a simple example, it is indicative of a wider problem. Even if we have taken great care to ensure that the expression representing a function is not susceptible to cancellation error we do not know whether the same can be said of the expression representing its derivative.

We can at least use interval arithmetic to monitor numerical error in the calculation of derivatives. You will recall that interval arithmetic works by placing bounds on arithmetic operations by computing the worst case smallest and largest results given the bounds on its arguments and numerical rounding.

The simple change to the expression object base class is illustrated in listing 14.

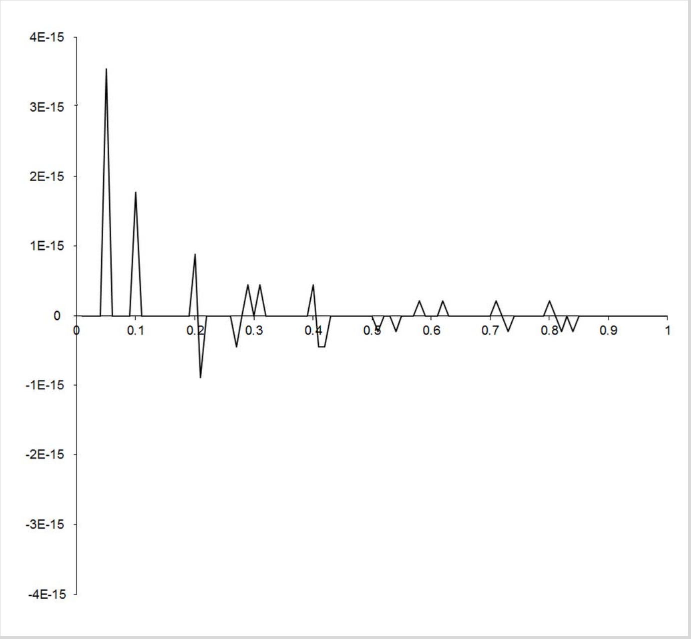


Figure 2

Figure 3 shows the result of evaluating the expression for the derivative of *x*/*x* using interval arithmetic and clearly reveals the presence of significant cancellation error near 0.

This combination of expression objects and interval arithmetic is as effective an algorithm for the calculation of derivatives as is possible without implementing a fully functional computer algebra system.

In practice, the principal drawback is in the expense of maintaining the expression tree and requiring virtual function calls to evaluate even simple arithmetic operations, although admittedly the latter can be largely avoided with the judicious use of templates.

Numerical approximation

There are many mathematical functions for which we have no simple formulae and must instead rely upon approximations. These

```
class expression_object
{
public:
  virtual ~expression_object() {}

  virtual interval   value() const = 0;
  virtual expression
     derivative(const expression &x) const = 0;
};
```

Listing 14

The derivative of an accurate approximating expression for a function is not necessarily an accurate approximating expression for the derivative of that function

approximations are often iterative in nature, stopping when some convergence criteria is satisfied.

If we naively implement an iterative approximation using an expression object as the argument we shall almost certainly run into trouble.

Not only is the resulting expression tree liable to be extremely large, it may take a different form for different values of the argument. If we generate an approximating expression using the initial value of the argument, we may very well introduce significant errors for other values.

To accurately represent iterative functions with expression objects we shall need to implement expression objects representing loops, conditional statements and so on and so forth.

Unfortunately working out the symbolic derivatives of such expressions is, in general, a tricky proposition.

Assuming that we have done so, or that the approximation takes the same form for all values of the argument, we still aren't quite out of the woods. The derivative of an accurate approximating expression for a function is not necessarily an accurate approximating expression for the derivative of that function.

For example, consider the function

$$g\left(x\right)=f\left(x\right)\times\left(1+\frac{\sin\left(100\times x\right)}{100}\right)$$

which serves as an approximation to *f*(*x*).

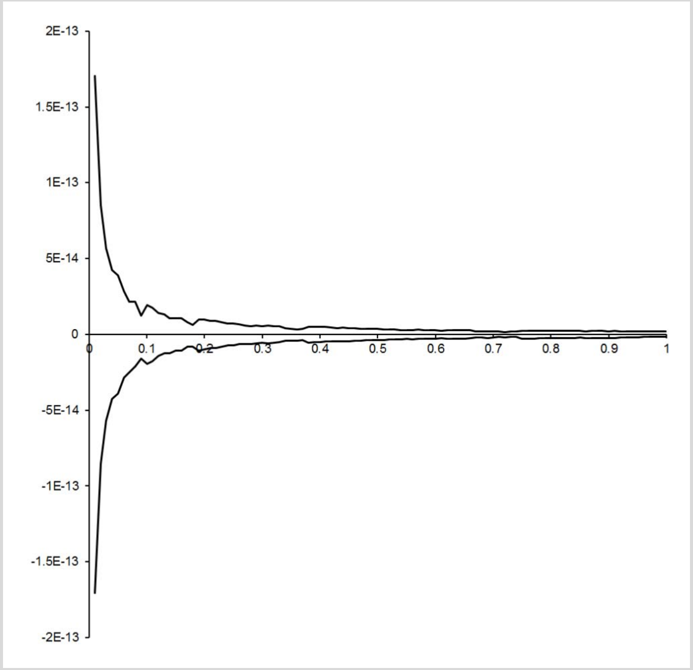


Figure 3

Now, this is trivially everywhere equal to within 1 percent of the value of *f*(*x*), but if we calculate its derivative we find that

$$\begin{aligned}\frac{d}{dx}g\left(x\right)&=\frac{d}{dx}f\left(x\right)\times\left(1+\frac{\sin\left(100\times x\right)}{100}\right)\\&\quad+f\left(x\right)\times100\times\frac{\cos\left(100\times x\right)}{100}\\&=\frac{d}{dx}f\left(x\right)\times\left(1+\frac{\sin\left(100\times x\right)}{100}\right)\\&\quad+f\left(x\right)\times\cos\left(100\times x\right)\end{aligned}$$

which can differ from the actual derivative by as much as 100 percent of the value of *f*(*x*)!

In such cases we may be better served by a polynomial approximation algorithm since these can effectively smooth out this high frequency noise by terminating once the derivative starts growing as δ shrinks close to its wavelength.

This observation hints at how we must proceed if we wish to use expression objects in conjunction with numerical approximations of functions.

Rather than represent the approximation of the function with an expression tree, we must create a new type of expression object that implements the approximation. The `derivative` member function must then return an expression object representing the derivative of the function.

In the worst case this may simply be an expression object implementing a polynomial approximation algorithm. Better still would be an expression object implementing a specifically designed approximation of the derivative. Best of all is the case when the derivative is known in closed form, such as the derivative of a numerical integration for example, in which case it can be an expression tree.

If necessary we can further implement an entire family of expression objects representing approximations of higher order derivatives to allow for higher orders of differentiation.

For these reasons I declare this approach to be a standing army of ducks; an effective fighting force, but rather expensive to feed and not always entirely welcome.

Quack two three four! Quack two three four! ■

References and Further Reading

[Harris11] Harris, R., *Overload*, Issue 102, ACCU, 2011.

[Ridders82] Ridders, C.J.F., *Advances in Engineering Software*, Volume 4, Number 2., Elsevier, 1982.

The Eternal Battle Against Redundancies, Part 2

Repeated information leads to poor quality software. Christoph Knabe continues to see how removing them has influenced language design.

Since the beginning of programming, redundancies in source code have prevented maintenance and reuse. By *redundancy* we mean that the same concept is expressed in several locations in the source code. Over the last 50 years the efforts to avoid redundancies [DRY] have inspired a large number of programming constructs. This relationship is often not obvious to programmers in their daily work. In part I [Part1] we talked about relative addressing, symbolic addressing, formula translation, parameterizable subroutines, control structures, middle-testing loops, symbolic constants, preprocessor features, and array initialization. In this part we will investigate higher concepts like object-oriented, aspect-oriented, and functional programming, as well as exception handling and even program generators and relational databases, and how these concepts contribute to redundancy avoidance. These concepts are discussed on the basis of prevalent programming languages. Whosoever understands the common concept is well equipped for the future.

Information hiding

The principle of information hiding was formulated by Parnas [Parnas]. It postulates not to allow direct manipulation of a data structure by clients. Such manipulations are to be done only through operations which are grouped in an interface. Information hiding was the prevalent design criterion in modular programming and it still plays an important role in object-oriented programming.

Enforcing the information hiding principle guarantees that the intended administration operations cannot be bypassed by a module's users. This contributes to redundancy avoidance by the fact that the logic behind the administrative functions cannot migrate into the user's code with the risk of duplication therein. This danger was always present in languages without support for information hiding.

Secure information hiding was enabled in C (1973) by the declaration of file-scope **`static`** variables. Such variables stayed alive beyond a function call, but were not accessible from outside the source file. Later languages which introduced special constructs for module interfaces and implementations were Modula-2 and Ada.

In C++ (1983) the information hiding principle was extended to user-defined data types (classes) by giving class members private visibility by default, which could be explicitly changed to **`public`**.

Genericity

COBOL (1960) had composite variables, but only Pascal (1970) introduced user-defined, composite data types as **`RECORD`**s. C (1973) followed with **`struct`**s. These constructs increased the robustness of programs, as confusions of e.g. persons with windows, calendar dates, or jobs were detected by the compiler. But the new strictness led to problems in the creation of universal services. Although Pascal had elegant operations for dynamic data structures, it was impossible to program a linked list so that it would be usable for an arbitrary element data type. The link data and the type of the payload data had to be firmly combined in the type for a list node. E.g.:

```
TYPE
  PersonList = ^PersonNode;
  PersonNode = RECORD
    info: Person;
    nextPtr: ^PersonNode;
  END;
```

If you wanted to use the same list management module in Pascal for different payload data types, you had to copy the source text and globally substitute the payload data type name.

The somehow less strict C could bypass such problems by using an untyped pointer, the **`void*`**. So in C it was possible, although insecure, to implement list management for arbitrary payload data. This list node could be formulated as follows:

```
struct List {
   void* infoPtr;
   struct List* nextPtr;
};
```

Only Ada (1980) achieved a synthesis of user-defined, composite data types (records) with flexible type safety. This concept was named genericity and was accepted by all modern, statically typed languages such as C++ (templates), Java, C#, and Scala. Using genericity you can avoid redundancies if you have to define same-behaviour services for different payload data types. The generic collection classes implemented by this technique are used quite frequently in all contemporary programming languages.

Dynamically typed languages such as Smalltalk or Ruby circumvent the problem described here by postponing the type checks to run-time.

Exception handling

In older programming languages (Lisp, Fortran, Algol, Cobol, Pascal, C) there was no automatic handling of exceptions. After every subroutine call the caller had to check manually whether the subroutine terminated successfully or erroneously. To complicate matters further there was no universal convention for how a subroutine should communicate its failure to the caller. The Unix services written in C used a special value for the function result as well as error codes in the global variable **`errno`**. The latter way was more suitable for standardization, as it did not have to cope with different function result types, but it was not suitable for the upcoming multi-threading.

How was the **`errno`** convention applied? After invoking **`fopen(filename, "r")`** in order to open a file you had to check whether **`errno`** had a nonzero value. As there were neither destructors nor garbage collection mechanisms in C, errors found could not be easily

Christoph Knabe learned programming at high school on a discarded Zuse 22, studied computer science from 1972, worked as a software developer at www.psi.de, and since 1990 has been professor of software engineering at the Beuth University of Applied Sciences Berlin (www.bht-berlin.de). Scala is the 14th language in which he has programmed intensively.

Inheritance alone enables a **minor avoidance of redundancy** by extracting common state and behavior of several data types into a base class

Mentioned Programming Languages

Note: The concepts are all talked about on the basis of prevalent programming languages. But often they were before tried out in research languages as Simula-67, CLU, MESA, or LISP dialects.

Name	Year	Innovations
Freiburgian Code	> 1958	Programming of the Zuse 22
Freiburgian Code Z23	1961	Relative and symbolic addressing
FORTRAN	1957	Formula translation, FORTRAN II: subroutines, linker
ALGOL	1958	Subroutines, block principle, BNF, control structures, recursion
LISP	1958	Garbage collection, recursion, functional programming (FP)
COBOL	1959	Record variables, long identifiers
Pascal	1970	Record types, pointer types, structured programming
Smalltalk	1972	Dissemination of object-oriented programming (OOP)
C	1973	Preprocessor, `sizeof`, operating system API, information hiding, `break`
Modula-2	1978	Separation of interface/implementation, `if...end`
Ada	1980	Genericity, automatic exception handling
C++	1983	Static typesafe OOP, freezing variable values, late declaration
Java	1995	Static typesafe OOP with garbage collection, stack trace API
AspectJ	2001	Centralized solution of cross-cutting concerns
Scala	2003	Static typesafe synthesis of OOP and FP

collected in working storage, so tended to be immediately reported. But this limited the universal usability of a subroutine, as then the destination of error reporting was not easily chosen by the caller.

So the correct handling of a function call in C on Unix, here of the function `fopen`, appeared as follows:

```
FILE* pFile = fopen(filename, "r");
if(errno!=0){
    perror(filename); //prints errno and filename
    fprintf(stderr, "at file %s in line %d\n",
       __FILE__, __LINE__);
    errno = FAILURE;
    return NULL;
}
```

You can easily imagine that correct error handling was highly redundant and made program texts harder to read and understand, and so harder to maintain. Furthermore, you had to write so much to implement this handling that programmers rarely practised it. Fortunately C's preprocessor macros offered a means to partially eliminate this redundancy. You could extract the portion of the example from `if` up to `return NULL;}` into a macro, which should get a context and the function result in case of failure as arguments.

```
#define ERRCHECK(context, failResult) ...
```

The invocation of `fopen` could then be much shorter:

```
FILE* pFile = fopen(filename, "r");
ERRCHECK(filename, NULL)
```

This approach cannot yet solve the problem of functions failing when they were combined in expressions, e.g. `f(x)*g(x)`. `ERRCHECK` could only be applied between two statements, not inside an expression.

Such error handling, which was implemented here manually, is done by contemporary languages automatically, when a function throws an exception. Standardized handling (usually a message with stack trace and program abortion) is guaranteed, although custom handling is possible. Automatic exception handling was popularized by Ada 80. C++ adopted it around 1990, while Java contained it from the beginning (including an API access to the stack trace of a caught exception).

Object-oriented programming

The technique of object-orientation, introduced by Simula 67 and popularized by Smalltalk-80, adopts 'information hiding' for object attributes and contains as innovations 'inheritance', 'reference polymorphism', and 'dynamic method dispatch'. Inheritance alone enables a minor avoidance of redundancy by extracting the common state and behaviour of several data types into a base class. Compared to composition this saves only a (relatively) small amount of writing when accessing an inherited attribute or method. Polymorphism of references enables a flexibility similar to the untyped pointers of C, but considerably more secure, as it constrains the referenced elements to subclasses of the base class. With dynamic dispatch for calls of virtual functions (C++, 1983) came the big, redundancy-avoiding progress, which is nowadays commonly known as the 'Template Method Pattern' [TemplMeth].

Template Method Pattern: As an example let us have a look at the problem of transaction management. In enterprise applications each operation of the business logic must be executed as a transaction. If the logic operation succeeds, the database modifications must be committed, otherwise errors must be reported and the database modifications must be rolled back. Instead of redundantly programming this behaviour in each logic method, you can extract it into an `execute` on a base class `Transaction`, which will call an abstract `action` method, which has to be overridden with the concrete logic operation. In Java, the solution looks like Listing 1.

The *template method* `execute` follows a fixed procedure in order to guarantee the `commit` or rollback. Only the business logic part of the action is conferred in the template method upon the abstract method `doAction`. The programmer of the subclasses has then to implement this method. Usage would follow the pattern shown below and would appear

Aspect-oriented programming enables you to handle concerns that cut across a software system

```
abstract class Transaction {
  public void execute(){
    final Connection con =
       DatabaseUtil.getConnection();
      try{
        doAction();
        con.commit();
      }catch(Exception ex){
        report(ex);
        con.rollback();
      }
  }
  abstract void doAction() throws Exception;
}
```

Listing 1

in a real system hundreds of times, which leads to an enormous reduction of redundancy, although the amount of code is still problematic.

```
new Transaction(){
  public void doAction() throws Exception {
    //Here the actual logic operation is placed.
  }
}.execute();
```

An alternative solution in Java would make use of reflection [Refl], as done by EJB 3.0 application servers internally. Each method of a class annotated as `@Session` is executed as a transaction.

Mixin Programming: In contrast to Java inheritance, Scala (2003) allows the mixing in of several *traits* (partially implemented interfaces), each of which can offer such template methods. The 'diamond problem' usually occurring with multiple inheritance is avoided by an explicitly definable resolution order. By this means you can freely combine different services in a class. In fact the Scala collections framework stands out due to an extremely high internal re-use of a few template methods. This is a big contribution to redundancy avoidance.

Aspect-oriented programming

Aspect-oriented programming enables you to handle concerns that cut across a software system centrally in an `aspect`. The above-mentioned problem of transaction management is exactly such a cross-cutting concern. Let us consider the case where each method of a logic façade should be executed as a transaction. Although the above solution, implementing the method `doAction` in an anonymous subclass of `Transaction`, is technically free of redundancy, it needs a lot of code. In contrast to this, in the solution with AspectJ (2001) in Listing 2, the aspect needs to be noted only once for the whole system. The 'pointcut' `executeAnyFacadeMethod` captures each execution of a method of objects of the type `LgFacade`. The `around` advice surrounds the captured method executions at the location, marked by `proceed`, thus causing the unified transaction management. This solution is not only technically, but also textually, free of redundancies. Usage of AspectJ in Java projects can deliver enormous redundancy savings straightaway.

```
aspect TransactionAspect {
  pointcut executeAnyFacadeMethod
      (LgFacade lgFacade):
     execution(public * *(..)) && this(lgFacade);

  Object around(LgFacade lgFacade):
     executeAnyFacadeMethod(lgFacade) {
    final Connection con =
       DatabaseUtil.getConnection();
    try{
      final Object result = proceed(lgFacade);
      con.commit();
      return result;
    }catch(Exception ex){
      report(ex);
      con.rollback();
    }
  }
}
```

Listing 2

Functional programming

Of the many and powerful constructs of Functional Programming I want to demonstrate only one, which facilitates the extraction of control structures. We take the every-day example that a list of persons should be displayed in a special format obtainable by method `getName` of class `Person`. In Java 5 we would need the function in Listing 3 to transform a list of persons into such a format.

A usage would look like:

```
personsToNames(persons)
```

The corresponding transformation in Scala would be so compact that no one would write a special function for this purpose:

```
persons.map(_.getName)
```

This is possible since the function `map` from the Scala collections library contains the above algorithm in a general solution and calls the argument function for each element of the `List`. Using the underscore sign `_` we define a mapping from an anonymous argument to the expression

```
public List<String> personsToNames
    (final List<Person> persons){
  final List<String> names =
     new LinkedList<String>();
  for(final Person p: persons){
    names.add(p.getName());
  }
  return names;
}
```

Listing 3

Sometimes an application needs **highly redundant code patterns**, but the programming language used does not offer a means to extract them

containing the underscore. The type of the argument is inferred from the element type of `persons` and thus needs not to be indicated explicitly.

In a similar way, in Scala you could guarantee the above-mentioned transaction management. What should be executed as transaction would have to be packed into `transaction{...}`, if the method `transaction` is suitably defined. This solution is technically free of redundancies, but it needs slightly more code than with AspectJ. In contrast, Scala needs only a minimum of keywords in comparison to AspectJ.

Program generators / domain specific languages

Sometimes an application needs highly redundant code patterns, but the programming language used does not offer a means to extract them. In such circumstances, as last resort, you could use a brute-force means: code generation. You define a special language, tailored to the problem, in which you can express yourself without redundancies. From that language you generate program code. Classical examples are decision table code generators like DETAB/65 or parser generators like *yacc*. As an example we give a rule of the contemporary parser generator ANTLR for multiplicative operations. This rule means: A product is a sequence of factors, which are separated by '*' or '/'.

```
product
    :    factor
         ( '*' factor
         | '/' factor
         )*
    ;
```

From this ANTLR can generate a parser which recognizes expressions like `a*b/c*d`. You can expand this parser to an interpreter or translator by inserting actions at the end of each line.

Data storage

Redundancies also cause problems in data storage. An example for this is a table of employees with the columns Id, Name, Date of Birth and Department.

Id	Name	Date of Birth	Department
1	Seyfried, Janina	17.01.1974	Human Resources
2	Stahl, Georg	06.06.1985	Sales
3	Schmidt, Sebastian	26.09.1979	Development
4	Müller, Friederike	19.11.1987	Sales

If the department is indicated as a string for each employee, this constitutes a data redundancy causing the following problems: If there is a typo in a department name, the affiliation of the employee to the department can not be recognized automatically. A renaming of a department necessitates modification of many employee rows.

The redundancy-free solution comprises the management of an additional table for departments, whose rows are referred to by a `departmentId` from each employee. Exactly this is achieved by normalization according to the concept of relational databases.

Other concepts of programming languages

This section lists relevant milestones in evolution of programming languages, which are not useful for redundancy avoidance, but are nevertheless worthy of mention.

- Robustness of programs was boosted by the declaration principle (Algol 58), by the locality helped by the block principle (Algol 58) and the late compilation in conjunction with a linker (FORTRAN, COBOL).
- Coding convenience was boosted by dynamically typed languages (LISP) or by the concedeclaration of variables only at their first usage (C++, 1983), by the freezing of computed values (C++), by 'Garbage Collection' instead of explicit deallocation (LISP, 1958).
- Labour division in development was boosted by the technique of separate pt of static type inference (Scala, 2003).
- Understandability was boosted by comments beginning with full line comments in FORTRAN with C, block comments in Algol with `comment` up to `;`, end of line comments in Ada with `--`, documentation comments in Java with `/**` up to nested block comments in Scala. COBOL pioneered long identifiers significantly helping understandability.

Summary

When you see how painfully the steps of progress in programming were achieved over the last 50 years, you really learn to appreciate the state of the art. Even more interesting is recognizing the driving force behind this progress. High redundancies in source code regularly required new programming constructs. In the majority of cases the ability was introduced to give a freely electable name to the redundant code pattern, and to invoke it with parameters from several locations. This happened to addresses, constants, subroutines, classes and generic units. Sometimes the evolution did not go as far and the redundant code patterns only received new keywords. This happened to formulas, loops, branches, and exception handling. When a programming language helped to eliminate redundancies better than a competing language, this was an advantage in the battle for dissemination. We can assume that this will still be true in future. ■

References and further reading

[DRY] http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[Parnas] http://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf

[Part1] Christoph Knabe: 'The Eternal Battle against Redundancies, Part I', *Overload* 106, December 2011, accu.org, pp. 6-10

[Refl] http://en.wikipedia.org/wiki/Reflection_%28computer_programming%29

[TemplMeth] http://en.wikipedia.org/wiki/Template_method_pattern

A Practical Introduction to Erlang

The future of massively parallel hardware will need good language support. Alexander Demin takes a look at an unexpected approach.

Since first hearing about functional programming, I have made many attempts to 'get it'. I felt it was something cool and worthwhile to learn, but I didn't make any real progress until…

For some reason my brain, spoiled by a decade of imperative programming, just didn't work this way. I was able to write a few snippets in Common Lisp and Scheme but I felt I couldn't write anything *real*. Racket was much better and I almost 'got' it, but this language seems overly complex to me. Haskell is still beyond me. But at some point I got a book called *Programming Erlang* by Joe Armstrong [Armstrong]. And at last, my journey into the functional world had begun.

To begin with, take a look at a quote from that book:

> A few years ago, when I had my research hat on, I was working with PlanetLab. I had access to the PlanetLab network, so I installed 'empty' Erlang servers on all the PlanetLab machines (about 450 of them). I didn't really know what I would do with the machines, so I just set up the server infrastructure to do something later.

That was cool I thought. If I had a cluster of 450 machines to play with, just imagine what could I do with this! Bare bones C or C++ don't give me much. Messing around with POSIX threads and TCP/IP directly will take ages to implement anything plausible from scratch. Even boostified C++ is lacking in this scenario. But this guy has an infrastructure out of the box!

So, what is Erlang? The language was developed at Ericsson to program telecoms devices. I expected it would be a system language like C, or at least Go if it had been created at that time. But fasten your seat belts – Erlang works in a managed environment (simply, a virtual machine executing a byte code), and secondly: it is a functional language. I was shocked – how can you deal with bits and bytes, packing and unpacking low-level network messages, process them efficiently under massive load and similar stuff, in a functional language?

Guys from Ericsson research were given the task of developing a language to build sophisticated but robust and scalable systems, and they had ended up with functional Erlang. This just got me.

After some time messing around with Erlang, I explored a few parallels with my everyday development in C++. I remember a John Carmack tweet with his sad story about a day spent finding a bug caused by a variable that had been accidentally changed somewhere. To avoid this waste of time, the variable should be simply declared as **`const`**. At some point I began to be obsessed by using **`const`** in C++. Increasing demand to write complex code safely led to my understanding that immutability is what I really want. In C and C++ that can be just putting **`const`** everywhere the logic of the code permits.

But Erlang takes immutability to another level. *All* variables in this language are immutable. Period. In fact, a variable can be assigned only once, when it gets created. Right after that it turns into a constant. Imagine in C or C++ that you must put **`const`** in front of any variable. Any mutation can be achieved only by copying and creating another variable. This is an extreme for C or C++, but this is the world of Erlang.

At first glance this looks like an obviously silly overhead. But let's slow down and give it a second thought. Yes, there is an overhead in copying, but at the same time the code has fewer side effects, and as a consequence fewer bugs caused by hidden data mutation in the complicated branching. Moreover, data is mutated only via copying, so the compiler and runtime (remember, Erlang is a managed environment) have many more clues on how to optimize and eliminate unnecessary copying. The runtime provides a set of native functions to mutate the data efficiently. For example, in Erlang, adding a new head to a list is quick but appending a tail is slow because it causes a deep copy. When you understand such peculiarities you can organize the mutations of data to be very efficient, yet free of side effects.

Finally, code without an internal state (and the immutability is a good guarantor of it) is much easier to parallelize in a multi-core or multi-machine environment, and Erlang has great support for multi-threading.

Well, I hope I've sown a seed of interest in Erlang, and it is time to code something real.

When I discover a new language, and after playing with trivial snippets, I often code a task called TCP/IP proxy. Simply, this application listens on a given TCP/IP port and for every incoming connection it connects to another remote host and passes the traffic back and forth between the caller and the remote host. Also the proxy logs all transmitted packets in the form of hexadecimal dumps. The application has to process multiple connections simultaneously.

This application can be very useful when you need, for example, to reverse engineer or debug an application protocol. From the implementation perspective it also very indicative – it involves string processing, multi-threading, sockets and file I/O.

In the past I've implemented it in C, C++, Python, PHP, Ruby and Go, and every time it was fun.

The code is below. Of course I cannot explain every single line if you're a newbie in Erlang, so you could check out the book *Programming Erlang* I mentioned above (at least the few first chapters to get basic concepts of Erlang).

I will go through the code and try to stress important elements. I recommend following and try to get a taste of Erlang.

Here we go. A file called `tcp_proxy.erl` (see Listing 1).

In lines 1 and 2 we define our unit of code, a module, and export one function **`main`** having one parameter – a list of command line arguments.

The definition of the function **`main`** (Listing 2) is similar in many other languages – if three command line arguments are supplied then this version is invoked.

```
1 -module(tcp_proxy).
2 -export([main/1]).
```

Listing 1

Alexander Demin Alexander is a software engineer holding a Ph.D. in Computer Science. He is constantly exploring new technologies and is always ready to drill down into the code with a disassembler to prove that the bug is there. He can be contacted at alexander@demin.ws

In Erlang the term 'process' means a different thing. It is a lightweight thread scheduled for execution not by the OS scheduler, but by the Erlang runtime

```
 3 main([ListenPort, RemoteHost, RemotePort]) ->
 4   ListenPortN = list_to_integer(ListenPort),
 5   RemotePortN = list_to_integer(RemotePort),
 6   start(ListenPortN, RemoteHost, RemotePortN);
```

Listing 2

```
 7 main(_) -> usage().
 8 usage() ->
 9   io:format("~n~s local_port remote_port
        remote_host~n~n", [?FILE]),
10   io:format("Example:~n~n"),
11   io:format("tcp_proxy.erl 50000 google.com
     80~n~n").
```

Listing 3

There's a second definition of the function **`main`** (Listing 3) which matches any other use that doesn't match the first. It simply prints usage information.

Now the fun begins in the function **`start`** (Listing 4). We create a TCP/IP listener (line 16) and then launch an acceptor thread (lines 19–21). A parallel thread (known as a process) is created by the Erlang **`spawn`** function.

At this point it is worth explaining the concept of processes in Erlang. Normally by 'a process' we mean an OS container having threads running within it. In turn 'a thread' usually means a single execution flow within a process planned for execution by the OS scheduler.

In Erlang the term 'process' means a different thing. It is a lightweight thread scheduled for execution not by the OS scheduler, but by the Erlang

```
12 start(ListenPort, CalleeHost, CalleePort) ->
13   io:format("Start listening on port ~p and
        forwarding data to ~s:~p~n",
14     [ListenPort, CalleeHost, CalleePort]),
15   ListenOptions = [binary, {packet, 0},
        {reuseaddr, true}, {active, true}],
16   case gen_tcp:listen(ListenPort,
        ListenOptions) of
17     {ok, ListenSocket} ->
18       io:format("Listener started ~s~n",
           [socket_info(ListenSocket)]),
19       spawn(fun() ->
20         acceptor(ListenSocket, CalleeHost,
             CalleePort, 0)
21       end),
22       receive _ -> void end;
23     {error, Reason} ->
24       io:format("Unable to start listener,
           error '~p'~n", [Reason])
25 end.
```

Listing 4

```
26 format_socket_info(Info) ->
27   {ok, {{A, B, C, D}, Port}} = Info,
28   lists:flatten(
        io_lib:format("~p.~p.~p.~p-~p",
        [A, B, C, D, Port])).

29 peer_info(Socket) ->
   format_socket_info(inet:peername(Socket)).
30 socket_info(Socket) ->
   format_socket_info(inet:sockname(Socket)).

31 format_date_time({{Y, M, D}, {H, MM, S}}) ->
32   lists:flatten(
33     io_lib:format("~4.10.0B.~2.10.0B.~2.10.
       0B-~2.10.0B.~2.10.0B.~2.10.0B",
34                   [Y, M, D, H, MM, S])).

35 format_duration({Days, {H, M, S}}) ->
36   lists:flatten(
37     io_lib:format(
         "~2.10.0B-~2.10.0B.~2.10.0B.~2.10.0B",
         [Days, H, M, S])).
```

Listing 5

runtime. It is possible to launch a few thousand processes in Erlang and the runtime will multiplex them onto native OS threads. Processes in Erlang are very fast to create and have a small memory footprint.

The concept of lightweight processes is quite similar to goroutines in Go (maybe goroutines were even inspired by Erlang). [Go] Moreover, the runtime can launch processes on remote Erlang nodes in exactly the same way as locally. Remember the quote from Joe's book at the beginning about 450 servers? This is where the magic begins.

From here I will use the term 'process' to refer to these lightweight parallel execution flows and not OS processes or threads.

In line 22 the main process starts receiving messages (we'll take a look at messaging a bit later). In fact nobody will be sending messages to the main process, so it will be blocked indefinitely unless you stop the application from outside.

Now the main process sleeps, but the acceptor process is ready to serve incoming connections.

In lines 26–37 (Listing 5) there are few string formatting routines. Nothing fancy.

Now the acceptor (Listing 6). In line 39 we accept an incoming connection, then in lines 41–43 launch another acceptor, and finally within the current process we connect to the remote host (line 44) and start forwarding data between the local and remote sockets by calling **`process_connection`** function (line 46).

I'd like you to pause and think. There is a pattern in many languages like C and C++ of how to implement a TCP/IP server: we have a main listener process; when an incoming connection comes up the listener creates a

```
38 acceptor(ListenSocket, RemoteHost,
     RemotePort, ConnN) ->
39   case gen_tcp:accept(ListenSocket) of
40     {ok, LocalSocket} ->
41       spawn(fun() ->
42         acceptor(ListenSocket, RemoteHost,
              RemotePort, ConnN + 1)
43       end),
44       case gen_tcp:connect(RemoteHost,
            RemotePort, [binary, {packet, 0}]) of
45         {ok, RemoteSocket} ->
46           process_connection(ConnN,
             LocalSocket, RemoteSocket);
47         {error, Reason} ->
48           io:format("Unable to connect to ~s:~s
             (error: '~p')",
49             [RemoteHost, RemotePort, Reason])
50         end;
51    {error, Reason} ->
52      io:format("Socket accept error '~w'~n",
          [Reason])
53   end.
```

Listing 6

worker process, passes the accepted socket to the worker for processing and continues to listen.

But our logic here is different: our listener processes the connection data within its own process because it also plays a worker role, but prior to the processing it clones itself to continue listening.

The former listener is now a worker. It processes the connection and then terminates.

This approach is natural for Erlang. It can be partially explained because in Erlang a process accepting a connection becomes its owner, and all messages from that connection will be delivered to the owner. This rule can be abused, but the point here is that processes are cheap and easy to create, and you're free to create as many as you want.

An Erlang developer usually fires up processes not per task, which tend to perform multiple activities, but per logically concurrent activity. And the activities don't need to multiplex anything.

Now we begin to process the connection (Listing 7). The **`process_connection`** function takes the number of the current connection and a pair of sockets. In line 57 it launches its own logger. Then via sending messages (line 59, 62 and 64) it communicates to it. By having the logger in a separate process we split the data transferring activity and the logger.

In lines 59 and 62 we send asynchronous messages but in the line 64 we send a synchronous one. At line 64, the processing of the connection is finished and we need to stop the logger by sending a **`stop`** command. But we must get an **`ack`** response back when the logger terminates (line 65).

Now the data transmitter (Listing 8). The **`pass_through`** function transmits data between two sockets and backs up the traffic to the logger.

```
54 process_connection(ConnN, LocalSocket,
     RemoteSocket) ->
55   LocalInfo = peer_info(LocalSocket),
56   RemoteInfo = peer_info(RemoteSocket),
57   Logger = start_connection_logger(ConnN,
        LocalInfo, RemoteInfo),
58   StartTime = calendar:local_time(),
59   Logger ! {connected, StartTime},
60   pass_through(LocalSocket, RemoteSocket,
        Logger, 0),
61   EndTime = calendar:local_time(),
62   Logger ! {finished, StartTime, EndTime},
63   % Stop the logger.
64   Logger ! {stop, self(), ack},
65   receive ack -> void end.
```

Listing 7

```
66 pass_through(LocalSocket, RemoteSocket,
     Logger, PacketN) ->
67   receive
68     {tcp, LocalSocket, Packet} ->
69       Logger ! {received, from_local, Packet,
            PacketN},
70       gen_tcp:send(RemoteSocket, Packet),
71       Logger ! {sent, to_remote, PacketN},
72       pass_through(LocalSocket, RemoteSocket,
            Logger, PacketN + 1);
73     {tcp, RemoteSocket, Packet} ->
74       Logger ! {received, from_remote, Packet,
            PacketN},
75       gen_tcp:send(LocalSocket, Packet),
76       Logger ! {sent, to_local, PacketN},
77       pass_through(LocalSocket, RemoteSocket,
            Logger, PacketN + 1);
78     {tcp_closed, RemoteSocket} ->
79       Logger ! {disconnected, from_remote};
80     {tcp_closed, LocalSocket} ->
81       Logger ! {disconnected, from_local}
82   end.
```

Listing 8

In line 67 we receive a next portion of the data from the sockets. Then in lines 68, 73, 78 and 80, using the pattern matching syntax of Erlang, we decide what kind of data has arrived and from where. The two branches starting at lines 69 and 74 mirror each other. Let's take a look at the first one. At line 69 it sends the received binary packet to the logger. Then it transmits the packet to the peer socket (line 70) and at line 71 it sends a notification message to the logger saying that the packet was delivered.

Take a look at line 72. This is a very important one. In Erlang there are no reserved words or operators for loops. You can simulate loops using lambdas and list comprehensions, but usually looping is implemented in Erlang by tail recursion. In line 72 the function **`pass_through`** calls *itself*. This doesn't mean that for every nested call it creates a new frame of the stack. Instead, the call to itself is the last one in the function execution flow, and the compiler optimizes such calls into tail recursion instead of normal recursion.

In short, tail recursion is a jump to the start of the calling function, but without using a proper context saving and restoring approach.

Finally in line 78 and 80 when we have the situation of a disconnected socket, the function doesn't call itself anymore and just exits.

The function **`start_connection_logger`** (Listing 9) begins a logging activity. It forms a name for the log and fires up a **`connection_logger`** function in a separate process. The function **`spawn_link`** differs from **`spawn`** by making the main process and a newly created one linked. If one dies or exits, its counterpart will be notified.

In Listing 10 (lines 92–95) we see four functions having the same name. To choose which function to call, Erlang applies the concept of pattern matching on the data, but not argument types like in C++ for instance. The key here is the first argument. In Erlang, identifiers starting with a lowercase letter are atoms. Atoms are implicit constants, enums if you like. When calling the **`peer_name`** function (lines 104, 112 and 124) if the first

```
83 start_connection_logger(ConnN, From, To) ->
84   {{Y, M, D}, {H, MM, S}} =
        calendar:local_time(),
85   LogName = lists:flatten(
86     % YYYY.MM.DD-hh.hh.ss-ConnN-From-To.log
87     io_lib:format(
         "log-~4.10.0B.~2.10.0B.~2.10.0B-"
88       "~2.10.0B.~2.10.0B.~2.10.0B-~4.10.0B-"
89       "~s-~s.log",
90       [Y, M, D, H, MM, S, ConnN, From, To])),
91   spawn_link(fun() -> connection_logger(ConnN,
        From, To, LogName) end).
```

Listing 9

```
92   peer_name(from_local, LocalInfo,
        _RemoteInfo) -> LocalInfo;
93   peer_name(from_remote, _LocalInfo,
        RemoteInfo) -> RemoteInfo;
94   peer_name(to_local, LocalInfo, _RemoteInfo)
        -> LocalInfo;
95   peer_name(to_remote, _LocalInfo, RemoteInfo)
        -> RemoteInfo.
```

Listing 10

argument is the `from_local` atom, for example, it calls the first version of the function.

Of course pattern matching is a much wider technique in Erlang, also used for branching. For example, given an expression you can provide a list of possible values expected in it and Erlang will try to match them all against the expression and pick a matching option. We'll see an example of this in a moment.

Again in lines 96 and 101 (Listings 11 and 12), we see two functions with the same name. The first one is called from `start_connection_logger`.

Take a closer look at the lines 97–99 (Listing 11). We create a lambda function called `Putter`. This lambda binds the two arguments function `append_message_to_file` to one argument lambda gluing the `LogName` variable as the second parameter.

The function `connection_logger` waits for an incoming message (line 102, Listing 12) and then does different type of logging activities depending on the received message, decided by pattern matching on the received data.

You may spot on that in lines 105, 113, 119, 125 and 132 we also create a lambda named `Message` and pass it to the `write_message` function. I will explain in a minute why we create and pass around a function rather than a string, for example.

The function `write_message` at the line 140 (Listing 13) calls the function passed by value in the `LogWriter` variable (remember the `Putter` at lines 97-99) and feeds it a variable called `Message` which also holds a function (remember the `Message` lambdas above). It then calls `connection_logger` again to get the next packet of data.

And this is a final bit – `append_message_to_file` function (line 143, Listing 14). This function again creates a lambda assigned to a `Printer` variable (line 145) and calls a function passed as a value in the `Putter` variable by passing the `Printer` as a parameter.

The whole idea of this multi-level cascade of lambdas is to split the putter activity formatting the data, and the printer activity writing the data into a log file and the console. The data formatting putter activity can be an expensive operation and should be done only once. That is why the `Putter` calls the `Printer` for every chunk of already formatted data, and the `Printer` in its turn outputs the data to the log file and the console (lines 146 and 147).

In lines 153 to 174 (Listing 15) we produce a nicely formatted hexadecimal dump. As we saw previously these conversion routines print out the formatted lines by calling a function passed in the `Printer` variable.

This is the end of the code.

Now it is time to try the application in action. You need to download and install the latest version of Erlang from http://www.erlang.org for your system. For Windows they ship pre-built bundles. I use a Mac and had to build Erlang from the source. It is a very straightforward procedure and worked for me without any problems.

To run our code you can try this:

```
escript tcp_proxy.erl 50000 pop.yandex.ru 110
```

```
 96   connection_logger(ConnN, From, To, LogName)
         when is_list(LogName) ->
 97     Putter = fun(Message) ->
 98       append_message_to_file(Message, LogName)
 99     end,
100     connection_logger(ConnN, From,
           To, Putter);
```

Listing 11

```
101 connection_logger(ConnN, FromInfo, ToInfo,
       LogWriter) ->
102   receive
103     {received, From, Packet, PacketN} ->
104       PeerName = peer_name(From, FromInfo,
             ToInfo),
105       Message = fun(Printer) ->
106         Printer("Received (#~p) ~p byte(s)
               from ~s~n",
107           [PacketN, byte_size(Packet),
             PeerName]),
108         binary_to_hex(Packet, Printer)
109       end,
110       write_message(ConnN, FromInfo, ToInfo,
             LogWriter, Message);
111     {sent, To, PacketN} ->
112       PeerName = peer_name(To, FromInfo,
             ToInfo),
113       Message = fun(Printer) ->
114         Printer("Sent (#~p) to ~s~n",
               [PacketN, PeerName])
115       end,
116       write_message(ConnN, FromInfo, ToInfo,
             LogWriter, Message);
117     {connected, Time} ->
118       When = format_date_time(Time),
119       Message = fun(Printer) ->
120         Printer("Connected to ~s at ~s~n",
               [ToInfo, When])
121       end,
122       write_message(ConnN, FromInfo, ToInfo,
             LogWriter, Message);
123     {disconnected, From} ->
124       PeerName = peer_name(From, FromInfo,
             ToInfo),
125       Message = fun(Printer) ->
126         Printer("Disconnected from ~s~n",
             [PeerName])
127       end,
128       write_message(ConnN, FromInfo, ToInfo,
             LogWriter, Message);
129     {finished, StartTime, EndTime} ->
130       Duration = calendar:time_difference
             (StartTime, EndTime),
131       When = format_date_time(EndTime),
132       Message = fun(Printer) ->
133         Printer("Finished at ~s,
              duration ~s~n",
134           [When, format_duration(Duration)])
135       end,
136       write_message(ConnN, FromInfo, ToInfo,
             LogWriter, Message);
137     {stop, CallerPid, Ack} ->
138       CallerPid ! Ack
139   end.
```

Listing 12

`escript` is one of the Erlang tools and should be in your path. It combines the compilation and execution phases together.

Once it gets started you type in a different window: `telnet localhost 50000` then when it gets connected type `QUIT` and press `ENTER`.

In the first window you should see something like Figure 1.

```
140 write_message(ConnN, FromInfo, ToInfo,
       LogWriter, Message) ->
141   LogWriter(Message),
142   connection_logger(ConnN, FromInfo, ToInfo,
         LogWriter).
```

Listing 13

```
append_message_to_file(Putter, LogName) ->
   {_, File} = file:open(LogName, [write,
      append]),
   Printer = fun(Format, Args) ->
     io:format(Format, Args),
     io:format(File, Format, Args)
   end,
   Putter(Printer),
   file:close(File).

% --------------------------------------------
```

Listing 14

That's it! It works.

Analyzing the code, if I implement such an application in C++ for instance, I always think twice before putting anything into a separate thread and so eventually end up with only a few threads – a listener, workers and one logger multiplexing logging from all the workers. Also I would think about a thread pool to limit the number of running native threads. Otherwise, accepting a thousand connections will spawn a thousand native threads which is not a clever approach even on a 32-core machine.

But in Erlang the multithreading is managed by the runtime. You can focus on the business logic of threading rather than on the OS resource management.

To conclude I'd like to underscore that I didn't want to explain every single character in the code. I touched very briefly on the fundamentals of Erlang such as pattern matching, messaging and list comprehensions. For better understanding I would recommend two books: *Programming Erlang: Software for a Concurrent World* [Armstrong] and *Erlang Programming* [Cesarini]. They perfectly complement each other.

I hope I have inspired someone to take a closer look at this wonderful language. There is a lot of stuff in there: passing function by values over the wire, hot swapping of code on live servers without restarting them, developing generic servers which can be turned to do anything (remember again that quote at the beginning), extending Erlang with your native code written in other languages and much more.

Have fun. ■

Source code

I've put the source from the article at [Github]. The version there is more advanced. As well as the text logger, it logs data in a binary form as well. It is not that exciting to study but is useful in real applications.

```
-define(WIDTH, 16).
binary_to_hex(Bin, Printer) ->
  binary_to_hex(Bin, Printer, 0).
binary_to_hex(<<Bin:?WIDTH/binary,
     Rest/binary>>, Printer, Offset) ->
  binary_to_dump_line(Bin, Printer, Offset),
  binary_to_hex(Rest, Printer,
     Offset + ?WIDTH);
binary_to_hex(Bin, Printer, Offset) ->
  Pad = fun() -> Printer("~*c",
     [(?WIDTH - byte_size(Bin)) * 3, 32]) end,
  binary_to_dump_line(Bin, Printer,
     Pad, Offset).
binary_to_dump_line(Bin, Printer, Offset) ->
  binary_to_dump_line(Bin, Printer, fun() ->
     ok end, Offset).
binary_to_dump_line(Bin, Printer,
    Pad, Offset) ->
  Printer("~4.16.0B: ", [Offset]),
  Printer("~s", [binary_to_hex_line(Bin)]),
  Pad(),
  Printer("| ~s~n",
     [binary_to_char_line(Bin)]).
binary_to_hex_line(Bin) ->
    [[(byte_to_hex(<<B>>) ++ " ") ||
    << B >> <= Bin]].
byte_to_hex(<< N1:4, N2:4 >>) ->
  [integer_to_list(N1, 16),
     integer_to_list(N2, 16)].
binary_to_char_line(Bin) ->
    [[mask_invisiable_chars(B) ||
    << B >> <= Bin]].
mask_invisiable_chars(X) when
    (X >= 32 andalso X < 128) -> X;
mask_invisiable_chars(_) -> $..
```

Listing 15

References

[Armstrong] *Programming Erlang: Software for a Concurrent World* by Joe Armstrong

[Cesarini] *Erlang Programming* by Francesco Cesarini and Simon Thompson

[Github] https://github.com/begoon/tcp_proxy/tree/logger_threads

[Go] http://golang.org/doc/effective_go.html#goroutines

```
Start listening on port 50000 and forwarding data to pop.yandex.ru:110
Listener started 0.0.0.0-50000
Connected to 93.158.134.37-110 at 2011.12.15-00.59.22
Received (#0) 38 byte(s) from 93.158.134.37-110
0000: 2B 4F 4B 20 50 4F 50 20 59 61 21 20 76 31 2E 30 | +OK POP Ya! v1.0
0010: 2E 30 6E 61 40 31 34 20 4D 78 55 55 33 74 66 48 | .0na@14 MxUU3tfH
0020: 52 57 32 31 0D 0A                               | RW21..
Sent (#0) to 127.0.0.1-51042
Received (#1) 6 byte(s) from 127.0.0.1-51042
0000: 51 55 49 54 0D 0A                               | QUIT..
Sent (#1) to 93.158.134.37-110
Received (#2) 20 byte(s) from 93.158.134.37-110
0000: 2B 4F 4B 20 73 68 75 74 74 69 6E 67 20 64 6F 77 | +OK shutting dow
0010: 6E 2E 0D 0A                                     | n...
Sent (#2) to 127.0.0.1-51042
Disconnected from 93.158.134.37-110
Finished at 2011.12.15-00.59.29, duration 00-00.00.07
```

Figure 1