Many slices of PI

Monte-carlo simulations in a parallel world

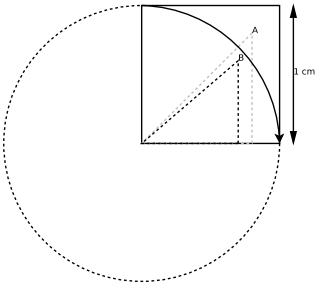
Steve Love, November 2011

Many slices of PI
Monte-carlo simulations in a parallel world
Steve Love, November 2011
All context copyright © Steve Love 2011

1. To be honest, Monte-Carlo simulations aren't really the focus of this talk. It's more about parallel programming. But since Monte-Carlo simulations usually benefit from lots of processing power, they're a natural candidate for parallelisation.
2. So to begin with, what is a Monte-Carlo simulation? In essence it's some algorithm that gets run lots of times (think for-loop) with the results aggregated in some way - usually averaging of some kind.
3. One simple application of this is to estimate PI, which can be done by repeatedly and randomly (with uniform distribution) dropping things inside a square and determining how many of them are inside a circle inscribed exactly within the square. The ratio of those within the circle to the total dropped is approximately PI / 4.

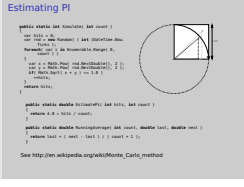
Estimating PI

```
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 x = Math.Pow( rnd.NextDouble(), 2 );
    var y = Math.Pow( rnd.NextDouble(), 2 );
    if( Math.Sqrt( x + y ) <= 1.0 )
      ++hits;
  }
  return hits;
}
```




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

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

See http://en.wikipedia.org/wiki/Monte_Carlo_method



1. The first listing is not the simulation, it's the method of estimating how many things are inside the circle. Also note that the ratio of 1/4 of a unit circle to a unit square is the same as the ratio for a circle of radius 0.5 to the same square.
2. The EstimatePi method calculates the required ratio and multiplies up by 4. The result of this, then, is an estimate of PI.
3. The third method is used to take a running average of the results. The idea of the whole thing is that the Simulate method gets called a lot!
4. These three methods are the ones used for *all* of the examples that follow.

Single slice

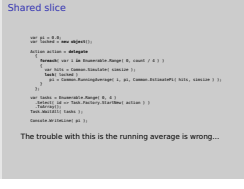
```
static void Main()
{
  var simsize = 999999;
  var count = 100;

  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 );
}
```


1. Here for the first time we have a real simulation. Single threaded, single process, for-loop (told you!)
2. Of course it's too slow. Provided we have access to a computer with more than one core or processor we can do better by writing a multi-threaded version.

Shared slice

```
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 );
```

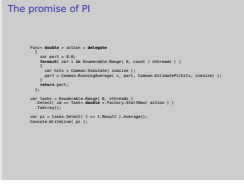
The trouble with this is the running average is wrong...



1. In this version there are 4 threads all working on a quarter of the total required. To make this version work each thread writes to a common result variable (pi) which must be synchronised.
2. It's important here to wait for all the results to be in before any attempt to use the pi variable.
3. The problem with the average is that part of calculating the *running* average requires the number of items so far, meaning that we'll really only calculate the average for 1/4 of the items...

The promise of PI

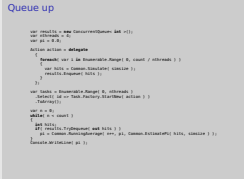
```
Func< double > action = delegate
  {
    var part = 0.0;
    foreach( var i in Enumerable.Range( 0, count / nthreads ) )
    {
      var hits = Common.Simulate( simsize );
      part = Common.RunningAverage( i, part, Common.EstimatePi(hits, simsize) );
    }
    return part;
  };

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

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

1. One solution to that problem is to use a Promise, whereby each thread performs the simulation and averaging of its quarter, then passes those results to a single collator to average *those* results giving a final average for the whole lot.
2. A great side-effect of this is that there is no longer a need for a lock! The use of t.Result means the calculation for pi will block until all tasks are complete. It's not exactly pretty though. Easily enough understood for this simple example, but 2 types of average?
3. There is a standard solution to the problem of needing a single consumer to gather results from multiple producers...

Queue up

```
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 );
```

1. Put the results on a queue. Sightly more verbose, but a lot more direct.
2. This version only enqueues the results of the simulation - how many things were inside the circle - instead of making any attempt to estimate PI.
3. A benefit here is that the collation of results can begin before all the tasks are done. The use of the queue (and waiting for it to drain) also means we have a natural synchronisation mechanism without using locks (or explicit joins).

Parallel PI

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

Action< int > action = delegate( int i )
  {
    var hits = Common.Simulate( simsize );
    results.Enqueue( hits );
  };

var options = new ParallelOptions { MaxDegreeOfParallelism = 4 };
Parallel.ForEach( Enumerable.Range( 0, count ), options, 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 );
```

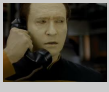

Parallel PI

1. Whilst we're here, in passing we note that the problem is embarrassingly parallel in the form we have now reached, so more starightforward parallel programming techniques can come into play. This is even more direct than the queue, and has all of its benefits.

“...but now we’d like to run the simulations on the grid...”

“!”

“...but now we’d like to run the simulations on the grid...”



“!”

1. No. It’s never that simple, is it? There’s always one...

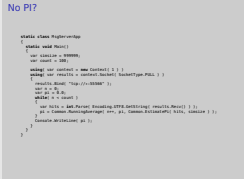
No PI?

```
static class MsgServerApp
{
  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 hits = int.Parse( Encoding.UTF8.GetString( results.Recv() ) );
        pi = Common.RunningAverage( n++, pi, Common.EstimatePi( hits, simsize ) );
      }
      Console.WriteLine( pi );
    }
  }
}
```

2011-11-27

└No PI?

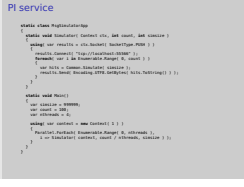
No PI?



1. Here's an application that takes the simulation results from...somewhere...and does the calculation to estimate PI. So, where is the simulation?

PI service

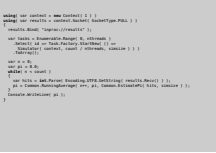
```
static class MsgSimulatorApp
{
  static void Simulator( Context ctx, int count, int simsize )
  {
    using( var results = ctx.Socket( SocketType.PUSH ) )
    {
      results.Connect( "tcp://localhost:55566" );
      foreach( var i in Enumerable.Range( 0, count ) )
      {
        var hits = Common.Simulate(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

1. Of course it's a service. In real life the configuration of the simsize, count, addresses and so on would be, er, configuration :-)
2. Note that this is a multi-threaded service. Each thread performs its own set of calcs and sends to the well-known end-point for further processing - indpendently of any other threads.
3. A tiny but measurable problem here is that the number of calculations must be a multiple of the number of threads to ensure that all of them are processed, otherwise further checks must be done to mop up any remainder.

Talkin' 'bout PI

```
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 hits = int.Parse( Encoding.UTF8.GetString( results.Recv() ) );
    pi = Common.RunningAverage( n++, pi, Common.EstimatePi( hits, simsize ) );
  }
  Console.WriteLine( pi );
}
```

1. With a simple change, we can use the simulation in-process, which has important implications for testing. This is the application code, which simply calls the Simulator as a thread...

Simulate PI

```
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() ) );
    }
  }
}
```


Simulate PI

1. ...and this is the service - now just a method.

In case you missed it...

Listing 1: In process PI

```
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() ) )
           ;
    }
  }
}
```

Listing 2: Out of proc PI

```
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() ) )
           ;
    }
  }
}
```

2011-11-27

└In case you missed it...

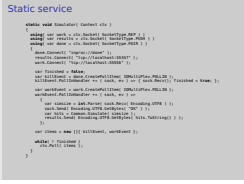
1. The difference between the implementations of the simulator service and the in-process version is subtle. Can you spot it?

Static service

```
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 );
  }
}
```

2011-11-27

└Static service



1. The previous service was multi-threaded, and we noted the issue about ensuring all the requested calculations are performed and reported.
2. A different approach is to just be a long-running server, and process *all* requests. Running multiple processes means that those processes can, potentially, be on different machines too - this is distributed parallelism.

Client

```
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 );
  }
}
```

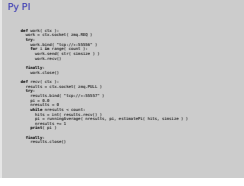
2011-11-27

└Client

1. Here's the client code for the distributed parallel version.
2. Most of the code is setting up the network side - the core of the program is still essentially the same as the original!
3. It doesn't end there. Using a suitably general middleware means...

Py PI

```
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()
```

1. ...you can even use a different programming language.
2. This is a client written in Python that will quite happily send work to, and collate results from the C# servers already shown.

Oh, the results

Style	PI	Time
single	3.14159434159434	24393.10
shared	3.1414178562569	6186.41
promise	3.14080278080278	6225.78
parallel	3.14156106156106	7199.58
msgqueue	3.14166458166458	6199.42
msgserver	3.14159398159398	6080.00
piclient	3.14153366153366	6525.49

piclient was run with 4 static servers in action.

Oh, the results

Style	PI	Time
single	3.14159434159434	24393.10
shared	3.1414178562569	6186.41
promise	3.14080278080278	6225.78
parallel	3.14156106156106	7199.58
msgqueue	3.14166458166458	6199.42
msgserver	3.14159398159398	6080.00
piclient	3.14153366153366	6525.49

piclient was run with 4 static servers in action.

1. These were actually run on a hyper-threaded 4 core box, but only with 4 threads in each case (including the parallel version using Parallel.ForEach). The measurements are only intended to be indicative of relative speeds - it was a simple Python script to launch each process and count system time until it finished.