The Multicore Revolution

Dr Russel Winder
It’z Interactive Ltd
russel@itzinteractive.com
The Multicore Revolution

We've Only Just Begun

Dr Russel Winder
It’z Interactive Ltd
russel@itzinteractive.com
“We’ve Only Just Begun”

by The Carpenters

will not be played.
Aims and Objectives

- Investigate some of the issues for software, programming, and programming languages raised by *The Multicore Revolution*.

- Educate (and hopefully entertain) people post Conference Dinner.
Structure of the Session

What sort of shape should this session have?
Structure of the Session

Astrid Jordana Byro

Rhombus-shaped

Symmetry group $D_2$

Structure of the Session

Kevlin Henney

Two dimensional with only four sides and regular? Knowing Russel, I would have assumed multi-faceted and somewhat irregular.
Structure of the Session

A rendering of the group E8.

http://www.aimath.org/E8/mcmullen.html
Structure of the Session

- Introduction.
- Do stuff.
- Exit stage left.
Structure of the Session

- Introduction.
- Do stuff.
- Exit stage left.

(Or right depending on one’s viewpoint.)
The Multicore Revolution
In the Beginning . . .
Core memory
... to About 5 Years Ago

http://verboten1.net/computers.jpg
Two core processors
1950 – 2005 (ish)
Moore’s Law

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

“Cramming more components onto integrated circuits”, Electronics Magazine 19 April 1965.
Moore’s Law

CPU Transistor Counts 1971-2008 & Moore’s Law

Curve shows ‘Moore’s Law’: transistor count doubling every two years

What to do?

- More cache
- More transistors
- Same processor
Speed as well as quantity . . .
It’s getting warm . . .

Faster and faster processors

Hotter and hotter computers
Too hot
It’s all too hot . . .

If we can’t run processors faster with more cache what do we do with all the transistors?

And remember there’ll be twice as many per chip very soon now.
More cores
We can market this!

If we have multiple cores then we can run them slower and hence cooler and yet execute many more instructions per second, so we can tell customers they are getting more bangs for their bucks.
Even if single-threaded programs run slower?
Talking of bangs and bucks
Large Hadron Collider – LHC

http://blogs.physicstoday.org/industry09/lhc.jpg
LHC

- Already 30 years old.
- 7TeV collisions of protons.
- £2.6bn + £575m.

A seriously big bang, although not the Big Bang.

Lots of bucks.
Why?
LHC – Why?

To find the Higgs Boson.
FUD

Don’t call it “The God Particle”.
The Standard Model – 1

SU(3) × SU(2) × SU(1)

Strong force | Weak force | Electro-weak force | Electromagnetics
The Standard Model – 2

http://en.wikipedia.org/wiki/Higgs_mechanism
The Standard Model – 3

- Particles have no mass in the Standard Model.
- Introduce the Higgs field.
- This gives mass to particles that interact with it.

François Engler and Robert Brout

Philip Anderson

Peter Higgs

Gerald Guralnik, C R Hagen, and Tom Kibble
Electro-weak ✔
Strong ✔
Gravity ✗

Use the Force.
We need a GUT.
The Multicore Revolution
1950 – 2005 (ish)
2005 (ish) – 2012 (ish)
Parallelism, not concurrency, is now the norm.
What is the Standard Model?

- High performance computing (HPC) is where the action has been:
  - Highly parallel computers.
  - Clusters of computers.

- HPC standard technology:
  - Fortran
  - MPI
  - OpenMP
Fortran is Cool (ish)

program pi
  implicit none
  integer , parameter :: n = 1000000000
  integer , parameter :: LongReal = selected_real_kind ( p = 18 )
  real ( LongReal ) , parameter :: delta = 1.0 / n
  real ( LongReal ) :: sum = 0.0 , elapseTime , pi_
  integer :: i , startTime , startFrequency , endTime , endFrequency
  call system_clock ( startTime , startFrequency )
  do i = 1 , n
    sum = sum + 1.0 / ( 1.0 + (( i - 0.5 ) * delta )**2 )
  end do
  pi_ = 4.0 * sum * delta
  call system_clock ( endTime , endFrequency )
  elapseTime = endTime - startTime
  elapseTime = elapseTime / startFrequency
  print *, "==== Fortran Sequential pi =", pi_
  print *, "==== Fortran Sequential iteration count =", n
  print *, "==== Fortran Sequential elapse =", elapseTime
end program pi
program pi
  use mpi
  implicit none
  integer, parameter :: n = 1000000000
  double precision, parameter :: delta = 1.0 / n
  double precision :: localSum = 0.0, sum = 0.0, elapseTime, pi_
  integer :: i, startTime, startFrequency, endTime, endFrequency
  integer :: errorState, nProcessors, myId, start, end, sliceSize
  call system_clock (startTime, startFrequency)
  call MPI_Init (errorState)
  call MPI_Comm_size (MPI_COMM_WORLD, nProcessors, errorState)
  call MPI_Comm_rank (MPI_COMM_WORLD, myId, errorState)
  sliceSize = n / nProcessors
  start = 1 + myId * sliceSize
  end = (myId + 1) * sliceSize
  do i = start, end
    localSum = localSum + 1.0 / (1.0 + ((i - 0.5) * delta)**2)
  end do
  call MPI_Reduce (localSum, sum, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD, errorState)
  call MPI_Finalize (errorState)
  if (myId == 0) then
    pi_ = 4.0 * sum * delta
    call system_clock (endTime, endFrequency)
    elapseTime = endTime - startTime
    elapseTime = elapseTime / startFrequency
    print *, "==== Fortran MPI pi =", pi_
    print *, "==== Fortran MPI iteration count =", n
    print *, "==== Fortran MPI elapse =", elapseTime
    print *, "==== Fortran MPI processor count =", nProcessors
  end if
end program
program pi
  implicit none
  integer , parameter :: n = 1000000000
  integer , parameter :: LongReal = selected_real_kind ( p = 18 )
  real ( LongReal ) , parameter :: delta = 1.0 / n
  real ( LongReal ) :: sum = 0.0 , elapseTime , pi_
  integer :: i , startTime , startFrequency , endTime , endFrequency , omp_get_num_procs
  call system_clock ( startTime , startFrequency )
  !$omp parallel do private ( i ) reduction ( + : sum )
  do i = 1 , n
    sum = sum + 1.0 / ( 1.0 + (( i - 0.5 ) * delta ) ** 2 )
  end do
  !$omp end parallel do
  pi_ = 4.0 * sum * delta
  call system_clock ( endTime , endFrequency )
  elapseTime = endTime - startTime
  elapseTime = elapseTime / startFrequency
  print * , "==== Fortran OpenMP pi =" , pi_
  print * , "==== Fortran OpenMP iteration count =" , n
  print * , "==== Fortran OpenMP elapse =" , elapseTime
  print * , "==== Fortran OpenMP processor count = " , omp_get_num_procs ( )
end program pi
Standard or Not?

- C has made inroads into the Fortran hegemony.
- C++ has also gained some traction.
- MPI and OpenMP still assumed, . . .
- . . . though Threading Building Blocks (TBB) makes C++ highly competitive compared to OpenMP.


```cpp
#include <iostream>
#include <iomanip>
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"
#include "microsecondTime.h"

class partialSum {
    private :
        long double delta ;
        long double sum ;
    public :
        partialSum ( const long double d ) : delta ( d ) , sum ( 0.0 ) {} 
        partialSum ( const partialSum & x , tbb::split ) : delta ( x.delta ) , sum ( 0.0 ) {} 
        void operator ( ) ( const tbb::blocked_range<long>& range ) {
            for ( long i = range.begin ( ) ; i != range.end ( ) ; ++i ) {
                const long double x = ( i - 0.5 ) * delta ;
                sum += 1.0 / ( 1.0 + x * x ) ;
            }
        }
        void join ( const partialSum & x ) { sum += x.sum ; }
        long double getSum ( ) { return sum ; }
};
```

---

**C++ and TBB – 1**

```cpp
#include <iostream>
#include <iomanip>
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"
#include "microsecondTime.h"

class partialSum {
    private :
        long double delta ;
        long double sum ;
    public :
        partialSum ( const long double d ) : delta ( d ) , sum ( 0.0 ) {} 
        partialSum ( const partialSum & x , tbb::split ) : delta ( x.delta ) , sum ( 0.0 ) {} 
        void operator ( ) ( const tbb::blocked_range<long>& range ) {
            for ( long i = range.begin ( ) ; i != range.end ( ) ; ++i ) {
                const long double x = ( i - 0.5 ) * delta ;
                sum += 1.0 / ( 1.0 + x * x ) ;
            }
        }
        void join ( const partialSum & x ) { sum += x.sum ; }
        long double getSum ( ) { return sum ; }
};
```
int main () {
    const long n = 1000000000l;
    const long double delta = 1.0 / n;
    const long long startTimeMicros = microsecondTime ( ) ;
    tbb::task_scheduler_init tbb_initializer;
    partialSum accumulator ( delta ) ;
    tbb::parallel_reduce ( tbb::blocked_range<long> ( 0 , n ) , accumulator , tbb::auto_partitioner ( ) ) ;
    const long double pi = 4.0 * accumulator.getSum ( ) * delta;
    const long double elapseTime = ( microsecondTime ( ) - startTimeMicros ) / 1e6 ;
    std::cout << "==== C++ TBB pi = " << std::setprecision ( 25 ) << pi << std::endl ;
    std::cout << "==== C++ TBB iteration count = " << n << std::endl ;
    std::cout << "==== C++ TBB elapse = " << elapseTime << std::endl ;
    return 0 ;
}
So what’s the problem?
2005 (ish) – 2012 (ish)
2005 (ish) – 2012 (ish)

Cache coherence.
Cache is . . .

- Level 1 – L1
- Level 2 – L2
- Level 3 – L3

http://static.commentcamarche.net/en.kioskea.net/faq/images/KjBt8WVH4Epro7ik.png
Cache is . . . the Problem

- Level 1 – L1
- Level 2 – L2
- Level 3 – L3

Just “sticking plaster” to avoid addressing the real issue.
1950 – 2005 (ish)
Moore’s Law

There will be more transistors so more cores, and more cache?
Possibly not.
There are indications that having more than 16 cores per memory leads to insurmountable problems.
Solution?
Change the model!
1950 – 2005 (ish)
Memory → Core → Interconnect → Core → Memory
Memory → Core → Interconnect → Core → Memory
Memory → Core → Interconnect → Core → Memory
Memory → Core → Interconnect → Core → Memory
This is not new.
There’s Nothing New . . .

- Standard model of a cluster system.
- Standard model of a blade system
- Standard model of all new multicore processors:
  - Tilera: Tile-64, Tile-100, . . .
  - Intel: 80-core (Poseidon), 48-core, . . .
Intel providing 48 x86_64 core processors!
2011 –
Interlude

- Edward Tufte wrote (an essay | a number of essays) lambasting the use of PowerPoint.
- The response by some wit was: “Every time you use PowerPoint, Edward Tufte kills a kitten.”
Interlude – Follow Up

- This is a OpenOffice.org presentation, so it doesn't count.

We will not have any kittens killed.
Can Current Languages Cope?

● Of course:
  - Fortran, C, C++ with OpenMP, MPI
  - Java – cf. blade server systems.
Can Current Languages Cope?

- Of course not:
  - No coherent programming model for the single program in Fortran, C, and C++.
    - SPMD isn’t likely to scale.
  - Vast amount of uniprocessor legacy code.
    - Obsession with auto-parallelization so as to avoid rewriting all the code from the 1950s and 1960s that every still uses.

Problem or opportunity?
1950 – 2005 (ish)
Solutions?

- Partitioned Global Address Space (PGAS)
  - Unified Parallel C
  - Co-array Fortran
  - Titanium
  - Chapel
  - X10
  - Fortress

The new Fortran standard is light years ahead of the new C++ standard.
**PGAS**

- Virtual address space is partitioned and mapped to the distributed memory of the underlying machine:
  - Chapel: domains
  - X10: places

Chapel – coforall

def execute ( param numberOfTasks : int ) {
    param n : int(64) = 1000000000 ;
    const delta : real = 1.0 / n ;
    param sliceSize : int(64) = n / numberOfTasks ;
    const eachProcessor : domain(1) = [ 0 .. ( numberOfTasks - 1 ) ] ;
    const results : [eachProcessor] real ;
    def partialSum ( const start : int(64) , const end : int(64), const delta : real ) : real {
        var sum : real = 0.0 ;
        for i in start .. end {
            sum += 1.0 / ( 1.0 + (( i - 0.5 ) * delta ) ** 2 ) ;
        }
        return sum ;
    }
    var timer : Timer ;
    timer.start ( ) ;
    coforall i in eachProcessor do results[i] = partialSum ( 1 + i * sliceSize , ( i + 1 ) * sliceSize , delta ) ;
    const pi = 4.0 * ( + reduce results ) * delta ;
    timer.stop ( ) ;
    writeln ( "==== Chapel Coforall pi = " , pi ) ;
    writeln ( "==== Chapel Coforall iteration count = " , n ) ;
    writeln ( "==== Chapel Coforall elapse = " , timer.elapsed ( ) ) ;
    writeln ( "==== Chapel Coforall task count = " , numberOfTasks ) ;
}
private static def execute ( numberOfTasks : int ) : void {
    val n : long = 1000000000l ;
    val delta : double = 1.0 / n ;
    val startTimeNanos : long = System.nanoTime ( ) ;
    val sliceSize : long = n / numberOfTasks ;
    val computeSlice = ( p : Point ) => {
        val id : int = p(0) ;
        val start : long = 1 + id * sliceSize ;
        val end : long = ( id + 1 ) * sliceSize ;
        var sum : double = 0.0 ;
        for ( var i : long = start ; i < end ; ++i ) {
            val x : double = ( i - 0.5 ) * delta ;
            sum += 1.0 / ( 1.0 + x * x ) ;
        }
        sum
    } ;
    val sums = Array.make[double] ( Dist.makeUnique ( ) , computeSlice ) ;
    val pi : double = 4.0 * sums.reduce ( Double.+ , 0.0 ) * delta ;
    val elapseTime : double = ( System.nanoTime ( ) - startTimeNanos ) / 1e9 ;
    Console.OUT.println ( "==== X10 Sequential pi = " + pi ) ;
    Console.OUT.println ( "==== X10 Sequential iteration count = " + n ) ;
    Console.OUT.println ( "==== X10 Sequential elapse = " + elapseTime ) ;
}
Executable Mathematics?

- Fortress

```fortress
export Executable
run () : () := do
  n = 100000  (* 10000 times fewer due to speed issues. *)
  d = 1.0 / n
  t_s = nanoTime ()

  \[ \pi_{est} = 4.0 \times \sum_{i=1:n} \left( 1.0 / (1.0 + ((i - 0.5) \times d)^2) \right) \]
  t_e = (nanoTime () - t_s) \times 10^{-9}
  print(“==== Fortress pi = “); println(\pi_{est})
  print(“==== Fortress iteration count = “); println(n)
  print(“==== Fortress elapse time = “); println(t_e)
end
```
Isn’t This All A Bit Experimental . . .

- . . . what to do today?
Solutions for Today

- Actor Model
- Dataflow Architectures
- Communicating Sequential Processes (CSP)

Return to a process view of applications.
It’s all about small processes passing messages to each other.
It is a different way of structuring software.
2011 –

![Diagram showing interconnect between memory and cores](image-url)
Make the programming model map easily to the operational semantics of the hardware.
GroovyCSP Example

```java
@Grab ( group = 'org.codehaus.jcsp' , module = 'jcsp' , version = '1.1-rc5-SNAPSHOT' )
@Grab ( group = 'org.codehaus.gpars' , module = 'gpars' , version = '0.10-beta-1-SNAPSHOT' )
import org.jcsp.lang.Channel
import org.jcsp.lang.CSProcess
import groovyx.gpars.csp.PAR
void execute ( final int numberOfTasks ) {
    final long n = 1000000000L
    final double delta = 1.0d / n
    final startTimeNanos = System.nanoTime()
    final long sliceSize = n / numberOfTasks
    final channels = Channel.one2oneArray ( numberOfTasks )
    final processes = []
    for ( int i = 0 ; i < numberOfTasks ; ++i ) { processes << new ProcessSlice_JCSP ( i , sliceSize , delta , channels[i].out() )}
    processes << new CSProcess () {
        public void run () {
            double sum = 0.0d
            for ( c in channels ) { sum += (double) c.in().read() }
            final double pi = 4.0d * sum * delta
            final double elapseTime = ( System.nanoTime () - startTimeNanos ) / 1e9
            System.out.println ( "==== Groovy/Java GPars CSP pi = " + pi )
            System.out.println ( "==== Groovy/Java GPars CSP iteration count = " + n )
            System.out.println ( "==== Groovy/Java GPars CSP elapse = " + elapseTime )
            System.out.println ( "==== Groovy/Java GPars CSP processor count = " + Runtime.getRuntime().availableProcessors () )
            System.out.println ( "==== Groovy/Java GPars CSP task count = " + numberOfTasks )
        }
    };
    ( new PAR ( processes ) ).run ()
}
```
Go from here and learn CSP.
Campaign for CSP libraries in Fortran, C and C++. 

*Java, Groovy and Python already have them!*
Think data parallel.
Data Parallelism

@Grab (group = 'org.codehaus.gpars', module = 'gpars', version = '0.10-beta-1-SNAPSHOT')
import groovyx.gpars.ParallelEnhancer
void execute (final int numberOfTasks) {
    final long n = 1000000000l
    final double delta = 1.0d / n
    final long startTimeNanos = System.nanoTime()
    final long sliceSize = n / numberOfTasks
    final items = [] ; for (i in 0..<numberOfTasks) { items << i }
    ParallelEnhancer.enhanceInstance (items)
    final pi = 4.0d * delta * items.collectParallel { taskId ->
        (new ProcessSlice(taskId, sliceSize, delta)).compute()}
}.sumParallel()
    final double elapseTime = (System.nanoTime() - startTimeNanos) / 1e9
System.out.println("==== Groovy/Java GPars ParallelEnhancer pi = " + pi)
System.out.println("==== Groovy/Java GPars ParallelEnhancer iteration count = " + n)
System.out.println("==== Groovy/Java GPars ParallelEnhancer elapse = " + elapseTime)
System.out.println("==== Groovy/Java GPars ParallelEnhancer processor count = " + Runtime.getRuntime().availableProcessors())
System.out.println("==== Groovy/Java GPars ParallelEnhancer task count = " + numberOfTasks)
Go from here and learn about parallel arrays.
Parallel arrays are already in Java (via jsr166y extras), Groovy (via GPars), and Scala (via Scalaz).

Even Fortran is getting them!

*C and C++ seem woefully inadequate in comparison.*

*Trapped in an MPI, OpenMP mindset?*
The Future of Software

- It's all about sequential process and message passing:

- Or data parallelism.
If you are writing applications and thinking in threads, then you are most likely part of the problem, not the solution.
Approaching the Close

Kevlin Henney

Two dimensional with only four sides and regular? Knowing Russel, I would have assumed multi-faceted and somewhat irregular.
Multicore hardware

Programming models

Programming languages

Quantum Chromodynamics
Summary – 1

- Uniprocessor thinking is now insufficient to the task of software development.
- Shared memory multi-threading is an operating system technique not an applications programming one.

Quarks and gluons are colourful.
Summary – 2

- Models such as:
  - Actor
  - Dataflow
  - CSP
  - Data parallelism

are the ones that applications should be structured with.

Quarks have flavour, gluons don’t.
Blatant Advertising

Python for Rookies
Sarah Mount, James Shuttleworth and Russel Winder
Thomson Learning Now called Cengage Learning.

Developing Java Software Third Edition
Russel Winder and Graham Roberts
Wiley

Buy these books!
Processor hardware is rushing off into the multicore future.
LHC is starting to study the Big Bang and other issues in “Big Physics”.
Software development needs a “big bang” to catch up with the multicore hardware revolution.
The Multicore Revolution

We've Only Just Begun

Dr Russel Winder
It’z Interactive Ltd
russel@itzinteractive.com
No cats were killed in the making of this presentation.
Except this one: