# Parallel Programming Concepts

# Software Programming Models - PGAS, Functional and Actor Programming

Peter Tröger

*Sources:*

*Martin Odersky, Lex Spoon, Bill Venners. Programming in Scala. Artima Press. 2008*
*Martin Odersky, Scala By Example. November 2009*
*Francesco Cesarini & Simon Thompson. Erlang Programming. O'Reilly. 2009*

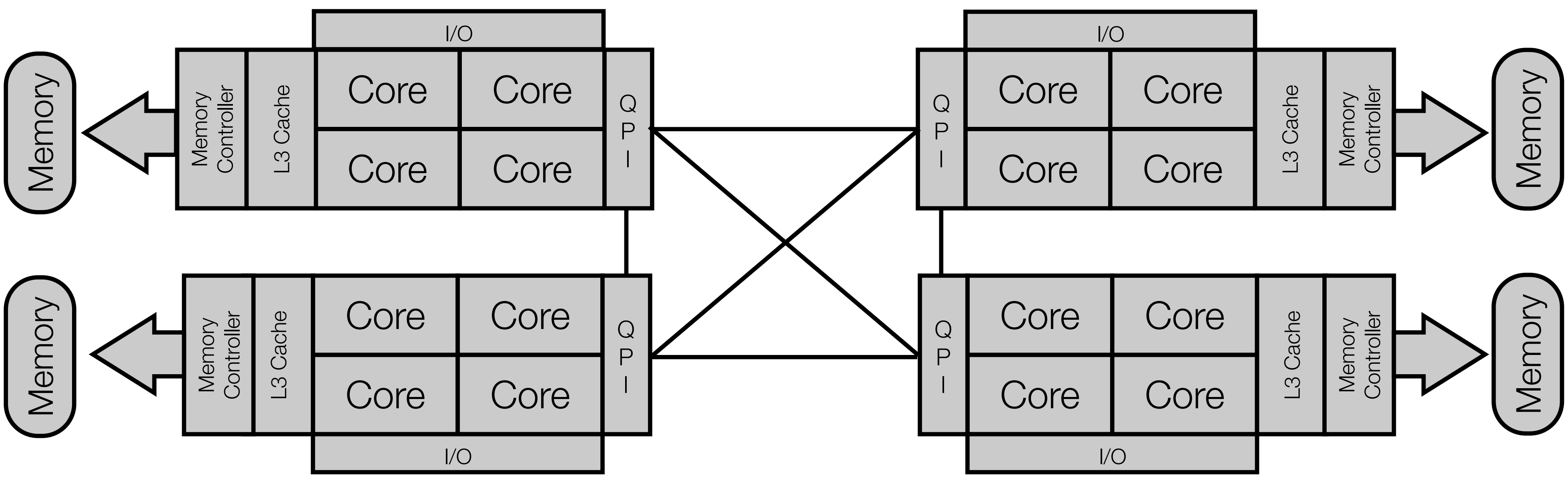
*Several Language Tutorials (see compiler web pages)*
*Functional Programming on Wikipedia*

# Programming Models

<table>
<tr><th></th><th>Task-Parallel Programming Model</th><th>Data-Parallel Programming Model</th><th>Actor Programming Model</th><th>Functional Programming Model</th><th>PGAS / DSM Programming Model</th></tr>
<tr><td>Shared Memory System</td><td>OpenMP, Threading Libs, Linda, Ada, Cilk</td><td>OpenMP, PLINQ, HPF</td><td rowspan="2">Scala, Erlang</td><td>Lisp, Clojure, Haskell, Scala, Erlang</td><td>-</td></tr>
<tr><td>Distributed Memory System</td><td colspan="2">Socket communication, MPI, PVM, JXTA, MapReduce, CSP channels</td><td>Scala, Erlang</td><td>-</td></tr>
<tr><td>Hybrid System</td><td>-</td><td>OpenCL</td><td>-</td><td>-</td><td>Unified Parallel C, Titanium, Fortress, X10,Chapel</td></tr>
</table>

# Traditional Parallel Programming

- Traditional approach:
  - Global shared memory, locks and explicit control flow
  - Mapped closely to hardware model of execution - so far
  - Flat shared memory model no longer fits to modern NUMA / GPU / MPP hardware development




-> **PGAS approaches**

# PGAS Approach

- Partitioned global address space (PGAS) approach for programming languages
  - Driven by high-performance computing community,
    as MPI + OpenMP alternative on large-scale SMP systems
  - Solves a real-world scalability issue, precondition for exa-scale computing
- Global shared memory, portioned into local parts per processor resp. activity
- Data is designated as local (near) or global (possibly far)
- PGAS language supports explicit access to remote data + synchronization
- Languages:
  - Unified Parallel C (Ansi C), Co-Array Fortran / Fortress (F90), Titanium (Java)
  - Chapel (Cray), X10 (IBM)
  - All under research, no wide-spread accepted solution on industry level

# Example: Unified Parallel C

- Extension of C for HPC on large-scale supercomputers (Berkeley)
- Considered by different HPC vendors (IBM, HP, Cray, ...)
- SPMD execution of UPC threads with flexible placement (MPI successor)
- Global shared address space among all (distributed) UPC threads
  - New qualifier ***shared*** to distinguish shared / non-shared UPC thread data
  - Shared data has affinity for a particular UPC thread
    - Primitive / pointer / aggregate types: Affinity with UPC thread 0
    - Array type: cyclic affinity per element, block-cyclic affinity, partitioning
- SPMD programming, MYTHREAD + THREADS variable

# Unified Parallel C

- Pointers
  - Pointers to `shared` data consist of thread ID, local address, and position
  - Pointer arithmetic supports blocked and non-blocked data distribution
- Loop parallelization with `upc_forall`
- No implicit assumptions on synchronization
  - `upc_lock, upc_unlock, upc_lock_attempt, upc_lock_t` (abstraction from implementation details)
  - `upc_barrier, upc_notify, upc_wait`

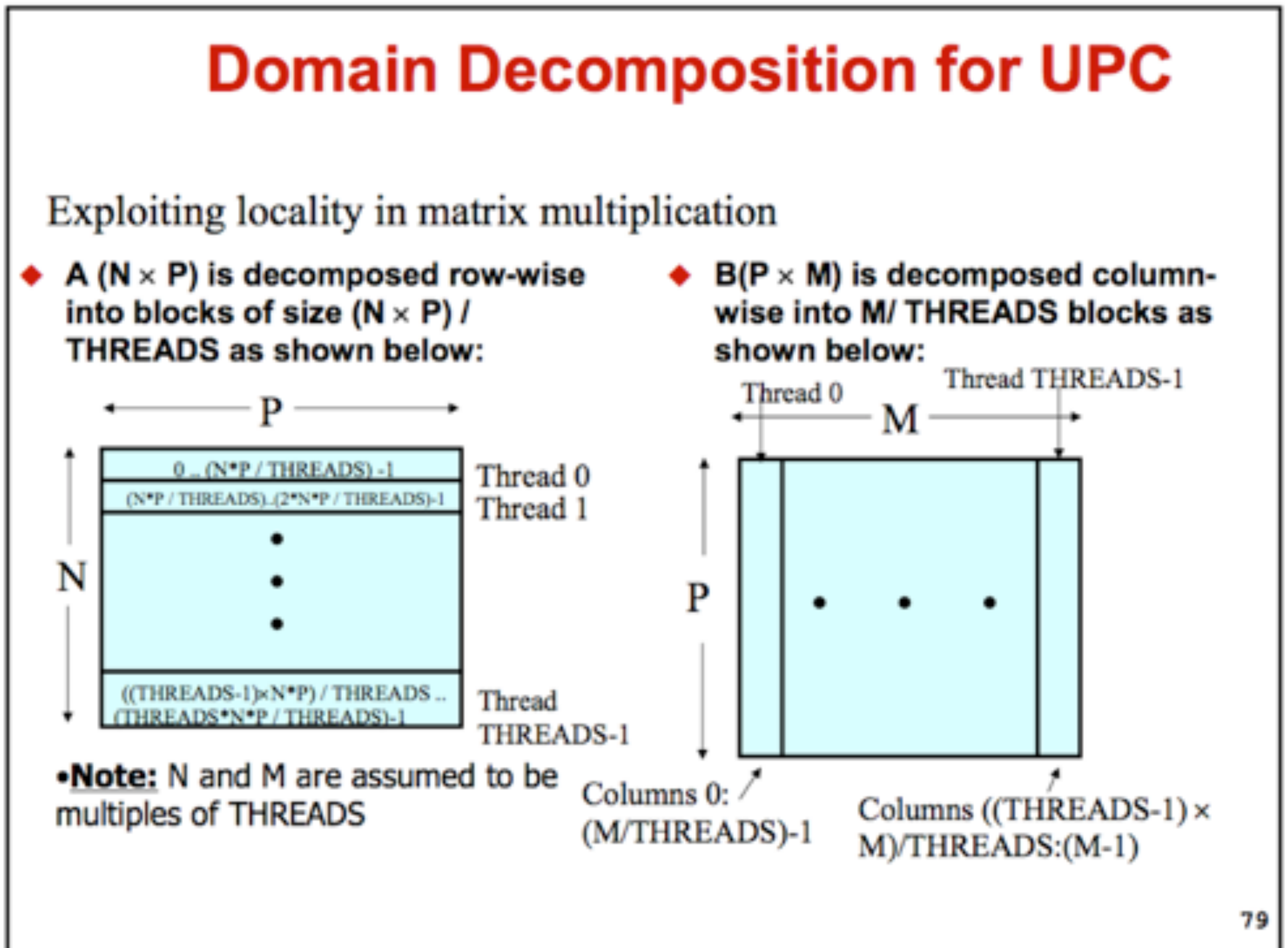
# Unified Parallel C

- Memory Consistency Model
  - Each memory reference / statement can be annotated accordingly
  - *Strict*: Sequential consistency (references from the same thread are in order)
  - *Relaxed*: Only issuing thread sees sequential consistency

```
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];
void main() {
  int i;
  for(i=0; i<N; i++)
  if (MYTHREAD==i%THREADS) v1plusv2[i]=v1[i]+v2[i];
}
```

# Unified Parallel C

- Still manual placement optimization needed, but data management is hidden




(C) Tarek El-Ghazawi

# Example: X10

- Parallel object-oriented PGAS language by IBM, research prototype
- Sequential X10 looks like extended version of Java (e.g. anonymous functions)
- Support for distributed cluster of SMP machines
- Java and C++ backends with according compilers, MPI support
- Fork-join execution model („async“), instead of SPMD approach in MPI

```
public class Fib {
  public static def fib(n:int) {
    if (n<=2) return 1;
    val f1:int;
    val f2:int;
    finish {
      async { f1 = fib(n-1); }
      f2 = fib(n-2);
    }
    return f1 + f2;
  }

  public static def main(args:Array[String](1)) {
    val n =
      (args.size > 0) ? int.parse(args(0)) : 10;
    Console.OUT.println("Computing Fib("+n+")");
    val f = fib(n);
    Console.OUT.println("Fib("+n+") = "+f);
  }
}
```

# X10 Concurrency

- Different parallel *activities*, each acting in one part of the address space (*Place*)
  - Direct variable access only in local place of the global address space
  - *Activities* are mapped to places, potentially on different machines
  - Application can perform blocking call to activity at another place:

    `val anInt = at(plc) computeAnInt();`

  - Fork parents can wait on child processes through `finish` clause
    - Childs cannot wait on parents (acyclic wait) - deadlock prevention

```
class HelloWholeWorld {
  public static def main(Array[String]):void {
     finish for (p in Place.places()) {
         async at (p) Console.OUT.println("Hello World from place "+p.id);
 }}}
```

# X10 Example: Parallel Sum

```
public class ParaSum {
  public static def main(argv:Rail[String]!) {
    val id = (i:Int) => i;   // integer identity function
    x10.io.Console.OUT.println("sum(i=1..10)i = " + sum(id, 1, 10));
    val sq = (i:Int) => i*i; // integer square function, inline def. used instead
    x10.io.Console.OUT.println("sum(i=1..10)i*i = " + sum((i:Int)=>i*i, 1, 10)); }

  public static def sum(f: (Int)=>Int, a:Int, b:Int):Int {
    val s = Rail.make[Int](1);
    s(0) = 0;
    finish {
      for(p in Place.places) {
        async{  // Spawn async at each place to compute its local range
          val pPartialSum = at(p) sumForPlace(f, a, b);
          atomic { s(0) += pPartialSum; }   // add partial sums
    }}}
    return s(0) }   // return total sum

  private static def sumForPlace(f: (Int)=>Int, a:Int, b:Int) {
    var accum : Int = 0;
    // each processor p of K computes f(a+p.id), f(a+p.id+K), f(a+p.id+2K), etc.
    for(var i : Int = here.id + a; i <= b; i += Place.places.length {
      accum += f(i); }
    return accum;
  }}
```

# Traditional Parallel Programming

- Imperative shared memory programming fails to solve concurrency issues
  - At each statement, developer must decide semantically upon locks to ensure correct data access and data modification
  - For each method call, one must reason about locks being held (deadlock)
  - Locks are not fixed at compile time, new might be created during run time
  - Additional locks might remove race conditions, but also add new deadlocks
  - -> Tackle the problem from a completely different direction
    - **Declarative programming** instead of imperative programming
    - **Message passing** instead of shared memory as concurrency base

# Declarative Programming Example - LINQ

- .NET „Language Integrated Query (LINQ)“
  - General purpose query facility, e.g. for databases or XML
  - Declarative standard query operators

```
var query = from p in products
            where p.Name.StartsWith("A")
            orderby p.ID
            select p;

foreach ( var p in query ) {
    Console.WriteLine ( p.Name );
}
```

- PLINQ is parallelizing the execution of LINQ queries on objects and XML data
- Declarative style of LINQ allows seamless transition to parallel version

```
IEnumerable<T> data = ...;
var q = data.Where(x => p(x)).Orderby(x => k(x)).Select(x => f(x));
foreach (var e in q) a(e);
```

```
IEnumerable<T> data = ...;
var q = data.AsParallel().Where(x => p(x)).Orderby(x => k(x)).Select(x => f(x));
foreach (var e in q) a(e);
```

# Functional Programming

- Programming paradigm that treats execution as function evaluation
  -> map some input to some output
  - Contrary to imperative programming that focuses on statement execution for global state changing (closer to hardware model of execution)
  - Programmer no longer specifies control flow explicitly
  - **Side-effect free computation** through avoidance of local state in functions
    -> enables **referential transparency** (no demand for particular control flow)
  - Typically strong focus on **immutable data** as language default
    -> instead of altering values, return altered copy
- Foundation Alonzo Church's lambda calculus from the 1930's
- First functional language was Lisp (late 50s), today Erlang, Haskell, Clojure, ...
- Trend to add functional programming features into imperative languages

# Imperative to Functional - Joel on Software

http://www.joelonsoftware.com/items/2006/08/01.html

```
alert("I'd like some Spaghetti!");
alert("I'd like some Chocolate Moose!");
```

```
function SwedishChef( food )
{
  alert("I'd like some " + food + "!");
}
SwedishChef("Spaghetti");
SwedishChef("Chocolate Moose");
```

```
alert("get the lobster");
PutInPot("lobster");
PutInPot("water");

alert("get the chicken");
BoomBoom("chicken");
BoomBoom("coconut");
```

```
function Cook( i1, i2, f ) {
  alert("get the " + i1);
  f(i1); f(i2); }

Cook( "lobster", "water",
       function(x) { alert("pot " + x); }  );
Cook( "chicken", "coconut",
       function(x) { alert("boom " + x); } );
```

# Imperative to Functional - Scala Example

```
def printArgs(args: Array[String]): Unit = {
  var i = 0
  while (i < args.length) {
    println(args(i))
    i+=1
}}
```

```
def printArgs(args: Array[String]): Unit = {
  args.foreach(println)
}
```

```
def formatArgs(args: Array[String]) =
  args.mkString(„\n")
```

# Imperative to Functional - Python

(C) David Merz, IBM

```
# Nested loop procedural style for finding big products
xs = (1,2,3,4)
ys = (10,15,3,22)
bigmuls = []
for x in xs:
    for y in ys:
        if x*y > 25:
            bigmuls.append((x,y))
print bigmuls
```

```
print [(x,y) for x in (1,2,3,4) for y in (10,15,3,22) if x*y > 25]
```

```
>>> student_tuples = [
        ('john', 'A', 15),
        ('jane', 'B', 12),
        ('dave', 'B', 10),
]
>>> sorted(student_tuples, key=lambda student: student[2])   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

# Functional Programming

- **Higher order functions**: Other functions as argument or return value
- **Pure functions**: No memory or I/O side effects
  - If the result of a pure expression is not used, it can be removed
  - A pure function called with side-effect free parameters has constant result
  - Without data dependencies, pure functions can run in parallel
  - A language with only pure function semantic can change evaluation order
  - Few functions with side effects (e.g. printing), typically do not return result
- Recursion as replacement for looping (e.g. factorial)
- Lazy evaluation possible, e.g. to support infinite data structures
- Why does this help with parallelism ? Think about Map / Reduce ...

# Example: Fortress (== „Secure Fortran“)

- Oracle (Sun) Programming Language Research Group, Guy L. Steele (Scheme, Common Lisp, Java)
- Language designed for (mathematical) high-performance computing
- Dynamic compilation, type inference
- Growable language: Prefer library over compiler
- Mathematical notation
    - Source code can be rendered in ASCII, Unicode, or as image
    - Greek letters, hundreds of operations
- Functional programming concepts, but also Scala / Haskell derivations

| | | | | | |
|---|---|---|---|---|---|
| BY | *becomes* | × | TIMES | *becomes* | × |
| DOT | *becomes* | · | CROSS | *becomes* | × |
| CUP | *becomes* | ∪ | CAP | *becomes* | ∩ |
| BOTTOM | *becomes* | ⊥ | TOP | *becomes* | ⊤ |
| SUM | *becomes* | ∑ | PROD | *becomes* | ∏ |
| INTEGRAL | *becomes* | ∫ | EMPTYSET | *becomes* | ∅ |
| SUBSET | *becomes* | ⊂ | NOTSUBSET | *becomes* | ⊄ |
| SUBSETEQ | *becomes* | ⊆ | NOTSUBSETEQ | *becomes* | ⊈ |
| EQUIV | *becomes* | ≡ | NOTEQUIV | *becomes* | ≢ |
| IN | *becomes* | ∈ | NOTIN | *becomes* | ∉ |
| LT | *becomes* | < | LE | *becomes* | ≤ |
| GT | *becomes* | > | GE | *becomes* | ≥ |
| EQ | *becomes* | = | NE | *becomes* | ≠ |
| AND | *becomes* | ∧ | OR | *becomes* | ∨ |
| NOT | *becomes* | ¬ | XOR | *becomes* | ⊕ |
| INF | *becomes* | ∞ | SQRT | *becomes* | √ |

```
<<Hello.fss>>=
component HelloWorld
    export Executable

    run()=do
        print "Hello, world!\n"
    end

end
```

# Fortress - Comparison to UPC

- No memory management, all handled by runtime system
- Implicit instead of explicit threading
- Set of types similar to C standard template library
- Fortress program state: Number of threads + memory
- Fortress program execution: Evaluation of expressions in all threads
- Component model integrated, import and export of interfaces
  - Components live in the ‚fortress' database, interaction through shell

# Fortress Syntax

- Adopt math whenever possible
  - Integer, naturals, rationals, complex number, floating point ...
  - Support for units and dimensions
- Everything is an expression, () is the void value
  - Statements are void-type expressions (while, for , assignment, binding)
  - Some statements have non-() values (if, do, try, case, spawn, ...)
  - `if x ≥ 0 then x else -x end`
  - `atomic x := max(x, y[k])`
- Generators: „`j:k`“ - range, „`j#n`“ - n consecutive integers from j, ...

# Fortress Basics

- *Object*: Fields and methods, *Traits*: Set of abstract / concrete methods
- Every object extends a set of traits

```
trait Boolean
  extends BooleanAlgebra⟦Boolean,∧,∨,¬,⊻,false,true⟧
  comprises { true, false }
 opr ∧(self, other: Boolean): Boolean
 opr ∨(self, other: Boolean): Boolean
 opr ¬(self): Boolean
end
object true extends Boolean
 opr ∧(self, other: Boolean) = other
 opr ∨(self, other: Boolean) = self
 opr ¬(self) = false
end
...
```

# Fortress - Functions

- Functions
  - Static (nat or int) parameters
  - One variable parameter
  - Optional return value type
  - Optional body expression
  - Result comes from evaluation of the function body

```
histogram⟦nat lo, nat sz⟧
          (a: A[#,#]): Int[lo#sz] =
do hist : Int[lo#sz] := 0
   for i,j ← a.indices do
      atomic do
         hist[a[i,j]] += 1
      end
   end
   hist
end
```

- `do-end` expression: Sequence of expressions with implicit parallel execution, last defining the blocks‘ result
  - Supports `also do` syntax for explicit parallelism

```
do
  factorial (10)
also do
  factorial (5)
also do
  factorial (2)
end
```

# Fortress - Parallelism

- Parallel programming as necessary compromise, not as primary goal
- Implicit parallelism wherever possible, supported by functional approach
  - Evaluated in parallel: function / method arguments, operator operands, tuple expressions (each element evaluated separately), loop iterations, sums
  - Loop iterations are parallelized

```
for i <- 1:5 do
  print(i "")
  print(i "")
end
```

```
for i <- sequential(1:5) do
  print(i "")
  print(i "")
end
```

  - Generators generate values in parallel, called functions run in parallel

Race condition handling through `atomic` keyword, explicit `spawn` keyword

*Ok, parallel code can be formulated*
*in a smarter way*
*by functional programming paradigms,*
*but what about*
*parallel execution coordination ?*

# Actor Model

- *Carl Hewitt, Peter Bishop and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence IJCAI 1973.*
  - Mathematical model for concurrent computation, inspired by lambda calculus, Simula, Smalltalk
  - No global system state concept (relationship to physics)
  - Actor as computation primitive, which can make local decisions, concurrently creates more actors, or concurrently sends / receives messages
  - Asynchronous one-way messaging with changing topology, no order guarantees
    - Comparison: CSP relies on hierarchy of combined parallel processes, while actors rely only on message passing paradigm only
  - Recipient is identified by *mailing address*, can be part of a message

# Example: Erlang

- Functional language with actor support, designed for large-scale concurrency
    - First version in 1986 by Joe Armstrong, Ericsson Labs
    - Released as open source since 1998
- Language goals from Ericsson product development demands
    - Scalable distributed execution with large number of concurrent activities
    - Fault-tolerant software under timing constraints
    - Online software update
- Applications:
  Amazon EC2 SimpleDB , Delicious, Facebook chat, T-Mobile SMS and authentication, Motorola call processing products, Ericsson GPRS and 3G mobile network products, CouchDB, EJabberD

# Erlang Language

- Sequential subset follows functional language approaches (strict evaluation, dynamic typing, first-class functions)
- Concurrency parts according to the actor model
- Control flow definition through pattern matching on set of equations:

```
area({square, Side})  -> Side * Side;
area({circle, Radius}) -> math:pi() * Radius * Radius.
```

- Atoms - constant literals, only comparison operation
- Lists and tuples are basis for complex data structures
- Single assignment variables, only call-by-value

# Sequential Erlang

- Influences by functional and logical programming (Prolog, ML, Haskell, ...)
- Control flow through conditional evaluation
  - CASE construct: Result is last expression evaluated on match

```
case cond-expression of
  pattern1 -> expr1, expr2, ...
  pattern2 -> expr1, expr2, ...
end
```

    - Catch-all clause not recommended here (‚defensive programming‘), since it might lead to match error at completely different code position
  - IF construct: Test until one of the guards evaluates to TRUE

```
if
  Guard1 -> expr1, expr2, ...
  Guard2 -> expr1, expr2, ...
end
```

# Concurrent Programming in Erlang

- Each concurrent activity is called *process,* only interaction through *message passing* - avoids typical shared memory issues (race conditions, *-locks)
- Designed for large number of concurrent activities (Joe Armstrong‘s tenets)
  - „The world is concurrent.“
  - „Things in the world don‘t share data.“
  - „Thins communicate with messages.“
  - „Things fail.“
- Design philosophy is to spawn a process for each new event
- Constant time to send a message
- `spawn(module, function, argumentlist) -` Spawn always succeeds, created process will eventually terminate with a runtime error (‚abnormally‘)

# Concurrent Programming in Erlang

- Communication via message passing, part of the language, no shared memory
  - Only messages from same process arrived in same order in the mailbox
- Send never fails, works asynchronously (`PID ! message`)
- Selective (not in-order) message retrieval from process mailbox
  - `receive` statement with set of clauses, pattern matching
  - If no clause matches, the subsequent mailbox content is matched
  - Process is suspended in receive operation until a match

```
receive
  Pattern1 when Guard1 -> expr1, expr2, ..., expr_n;
  Pattern2 when Guard2 -> expr1, expr2, ..., expr_n;
  Other   -> expr1, expr2, ..., expr_n
end
```

# Erlang Example

```erlang
% Create a process and invoke the function web:start_server(Port, MaxConnections)
ServerProcess = spawn(web, start_server, [Port, MaxConnections]),

% Create a remote process and invoke the function
% web:start_server(Port, MaxConnections) on machine RemoteNode
RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConnections]),

% Send a message to ServerProcess (asynchronously). The message consists of a tuple
% with the atom "pause" and the number "10".
ServerProcess ! {pause, 10},

% Receive messages sent to this process
receive
        a_message -> do_something;
        {data, DataContent} -> handle(DataContent);
        {hello, Text} -> io:format("Got hello message: ~s", [Text]);
        {goodbye, Text} -> io:format("Got goodbye message: ~s", [Text])
end.
```

(C) Wikipedia

# Concurrent Programming in Erlang

- Processes can be registered with Pid under a name (see shell „`regs().`“)
  - Registered processes are expected to provide a stable service
  - Messages to non-existent processes under alias results in caller error
- Timeout for receive through additional `after` block

```
receive
  Pattern1 when Guard1 -> expr1, expr2, ..., expr_n;
  Pattern2 when Guard2 -> expr1, expr2, ..., expr_n;
  Other   -> expr1, expr2, ..., expr_n
after
  Timeout -> expr1, expr2, ...
end
```

- Typical process pattern: Spawned, register alias, initialize local state, enter receiver loop with current state, finalize on some stop message

# Concurrent Programming in Erlang

- Receiver loop typically modeled with tail-recursive call
  - Receive message, handle it, recursively call yourself
  - Tail recursion ensures constant memory consumption
- Non-handled messages in the mailbox should be considered as bug, avoid defensive programming approach (‚throw away without notice‘)
- Messaging deadlocks are easily preventable by considering the *circular wait* condition
- Libraries and templates available for most common design patterns
  - Client / Server model - clients access resources and services
  - Finite state machine - perform state changes on received message
  - Event handler - receive messages of specific type

# Example: Tail-Recursion, Read-Only Variables

```
loop(Module, State) ->
receive
{call, From, Request} ->
  {Result, State2} = Module:handle_call(Request, State),
  From ! {Module, Result},
  loop(Module, State2);
{cast, Request} ->
  State2 = Module:handle_cast(Request, State),
  loop(Module, State2)
end.
```

- For unchanged parameters at the same position, no byte code is generated
- Subroutine call turns into a jump
- No new stack frame per call

# Erlang Robustness

- In massively concurrent systems, you don‘t want implicit process dependencies -> Message passing and `spawn` always succeed
- Generic library modules with in-built robustness (e.g. state machines)
- Race conditions are prevented by *selective receive* approach
  - Messages are not processed in order, but based on match only
  - Good for collecting responses for further processing, or rendevous
  - Transfer of PID supports data sharing by copy with unknown partners

```
PidB!{data, self()}  receive
                       {data, PidA}->PidA!response(data)
                     end
```

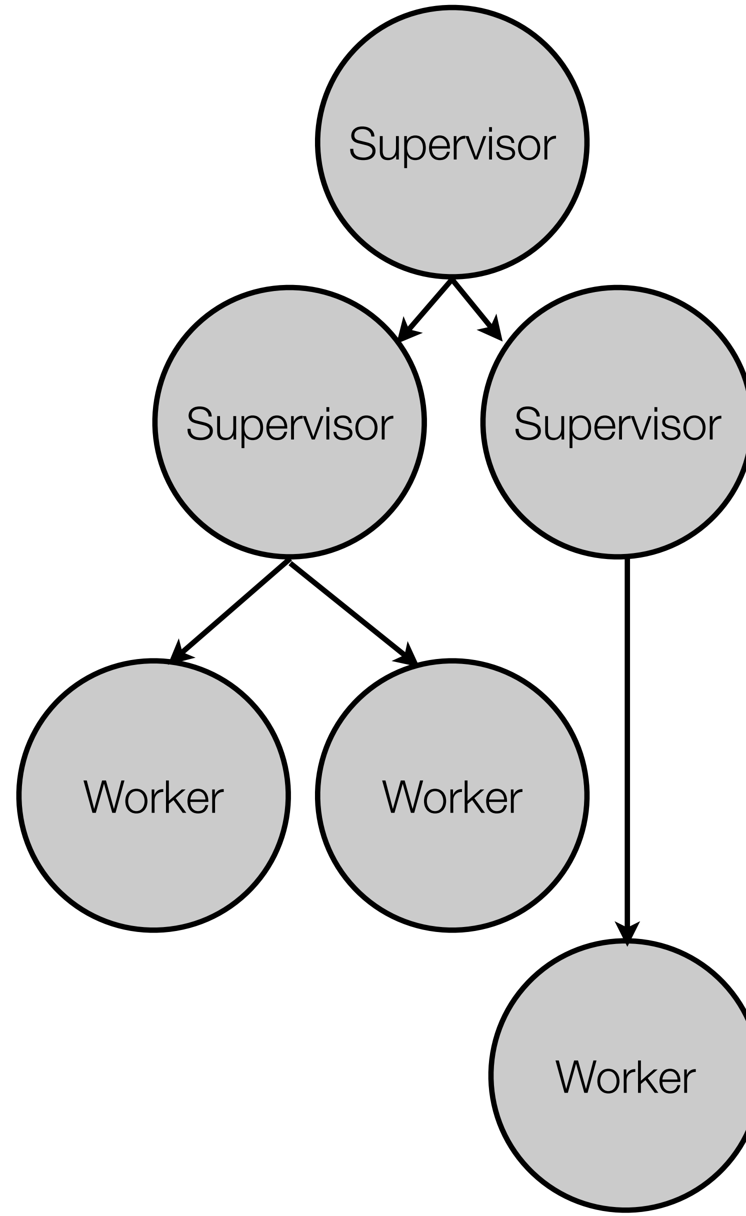
# Erlang Robustness

- Credo: „Let it crash and let someone else deal with it“, „crash early“
- In-build function `link()` creates bidirectional link to another process
  - If a linked process terminates abnormally, *exit signal* is sent to buddies
  - On reception, they send *exit signal* to their linked partners, containing the same `reason` attribute, and terminate themselves
- Processes can trap incoming exit signals through configuration, leading to normal message in the inbox
- Unidirectional variant `monitor()` for one-way surveillance
- Race conditions still can occur, standard build-in atomic function available

```
link(Pid = Spawn(Module, Function, Args))
Pid = spawn_link(Module, Function, Args)
```

# Erlang Robustness

- Robustness through layering in the process tree
  - Leave processes act as worker (application layer)
  - Interior processes act as supervisor (monitoring layer)
  - Supervisor shall isolate crashed workers from higher system layers through exit trap
  - Rule of thumb: Processes should always be part of a supervision tree
  - Allows killing of processes with updated implementation as a whole -> HA features



# Example: Scala - „Scalable Language“

- Martin Odersky, École Polytechnique Fédérale de Lausanne (EPFL)
- Combination of OO- and functional language features
  - Expressions, statements, blocks as in Java
  - Every value is an object, every operation is a method call
  - Classes and traits, objects constructed by *mixin-based composition*
  - Implicit conversions for objects
  - Functions are first-class values
- Most language constructs are library functions, can be overloaded
- Compiles to JVM byte code, interacts with Java libraries, re-use of types
- Use case: Twitter moved from Ruby to Scala in 2009

# Scala - Quicksort

```
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i)
    xs(i) = xs(j); xs(j) = t; ()
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1; j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

- Similar to standard imperative languages
- Functions in functions, global variables
- Read-only value definition
- Every function returns a result (expression-oriented language)
  - `Unit` / `()` return value for procedures

# Scala - Quicksort

```
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
}}
```

- Functional style (same complexity, higher memory consumption)
  - Return empty / single element array as already sorted
  - Partition array elements according to pivot element
    - Higher-order function *filter* takes *predicate function* („pivot > x“) as argument
  - Sort sub-arrays accordingly

# Scala - Operators are Methods

```
val sum = 1 + 2
```

```
val sum = (1).+(2)
```

Operator overloading

```
val longSum = 1 + 2L
```

Infix operators

```
s indexOf 'o'
s indexOf ('o', 5)
xs filter (pivot >)
```

Implicit conversion to rich wrappers

```
0 max 5
4 to 6
"bob" capitalize
```

# Scala - Object-Oriented Programming

```
class Rational(n: Int, d: Int) {
  require (d != 0)
  val numer: Int = n
  val denom: Int = d
  override def toString = numer + „/“ + denom
  def this(n: Int) = this(n, 1)
  def *(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom )
  def *(i: Int): Rational =
    new Rational(numer*i, denom)
}
```

# Scala - Functions

- Functions as first-class value - pass as parameter, use as result

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)

def sumInts(a: Int, b: Int): Int = sum(id, a, b)

def id(x: Int): Int = x

def sumSquares(a: Int, b: Int): Int = sum(square, a, b)

def square(x: Int): Int = x * x
```

- Anonymous functions

```
def sumSquares(a: Int, b: Int): Int =
  sum((x: Int) => x * x, a, b)
```

# Scala - Functions

- Parameter type deduction

```
def sumSquares(a: Int, b: Int): Int =
  sum((x: Int) => x * x, a, b)

def sumSquares(a: Int, b: Int): Int =
  sum(x => x * x, a, b)
```

- Currying - Transform multiple parameter function into chain of functions

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)

def sum(f: Int => Int): (Int, Int) => Int = {
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)
    sumF
}
def sumSquares = sum(x => x * x)     „scala> sumSquares(1, 10)"
```

# Scala - Case Classes

```
abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

- Case classes have (1) an implicit constructor, (2) accessor methods for constructor arguments, and (3) implementations of `toString, equals, hashCode`

  ```
  Sum(Sum(Number(1), Number(2)), Number(3))
  ```

- Foundation for pattern matching - generalized `switch` statement

  ```
  def eval(e: Expr): Int = e match {
    case Number(n) => n
    case Sum(l, r) => eval(l) + eval(r)
  ```

Pattern

Expression

# Scala - Program Execution as Substitution

```
eval(Sum(Number(1), Number(2)))
...
Sum(Number(1), Number(2)) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(n1) + eval(n2) }
...
eval(Number(1)) + eval(Number(2))
...
Number(1) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(n1) + eval(n2)
} + eval(Number(2))
...
1 + eval(Number(2))
...
1+2
...
3
```

# Scala - Functional Programming Support

- Functional objects
  - Do not have any mutable state
- Collection libraries differentiate between mutable and immutable classes
  - *Arrays* vs. *Lists*
  - Two different sub-traits for *Set* type, differentiation by name space
  - Immutable version of collection as default

```
import scala.collection.mutable.Set
val movieSet = Set(„Hitch“, „Poltergeist“)
movieSet += „Shrek“
println(movieSet)
```

# Scala - Concurrent Programming Tools

- Implicit superclass is `scala.AnyRef`, provides typical monitor functions

```
scala> classOf[AnyRef].getMethods.foreach(println)
def wait()
def wait(msec: Long)
def notify()
def notifyAll()
```

- `Synchronized` function, argument expression is executed mutually exclusive

```
def synchronized[A] (e: => A): A
```

- Synchronized variable with `put`, blocking `get` and invalidating `unset`

```
val v=new scala.concurrent.SyncVar()
```

- Futures, reader / writer locks, semaphores, mailboxes, ...

```
import scala.concurrent.ops._
...
val x = future(someLengthyComputation)
anotherLengthyComputation
val y = f(x()) + g(x())
```

- Explicit parallelism through `spawn (expr)`

# Scala - Concurrent Programming

- Actor-based concurrent programming, as introduced by Erlang
  - Concurrency abstraction on-top-of threads
  - Communication by asynchronous sends and synchronous receive blocks

```
actor {
  var sum = 0
  loop {
    receive {
      case Data(bytes)       => sum += hash(bytes)
      case GetSum(requester) => requester ! sum
}}}
```

  - All constructs are not part of the language implementation, but library functions `(actor, loop, receiver, !)`
  - Alternative `self.receiveWithin()` call with timeout

# Scala - Concurrent Programming

```scala
class Pong extends Actor {
  def act() {
    var pongCount = 0
    while (true) {
      receive {
        case Ping =>
          if (pongCount % 1000 == 0)
            Console.println("Pong: ping "+pongCou
          sender ! Pong
          pongCount = pongCount + 1
        case Stop =>
          Console.println("Pong: stop")
          exit()
      }
    }
  }
}
```

```scala
object pingpong extends Application {
  val pong = new Pong
  val ping = new Ping(100000, pong)
  ping.start
  pong.start
}
```

```scala
class Ping(count: int, pong: Actor) extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! Ping
    while (true) {
      receive {
        case Pong =>
          if (pingsLeft % 1000 == 0)
            Console.println("Ping: pong")
          if (pingsLeft > 0) {
            pong ! Ping
            pingsLeft -= 1
          } else {
            Console.println("Ping: stop")
            pong ! Stop
            exit()
          }
      }
    }
  }
}
```

# Scala - Actor Case Classes

```
import scala.actors.Actor

abstract class AuctionMessage
case class Offer(bid: Int, client: Actor) extends AuctionMessage
case class Inquire(client: Actor) extends AuctionMessage

abstract class AuctionReply
case class Status(asked: Int, expire: Date) extends AuctionReply
case object BestOffer extends AuctionReply
case class BeatenOffer(maxBid: Int) extends AuctionReply
case class AuctionConcluded(seller: Actor, client: Actor) extends AuctionReply
case object AuctionFailed extends AuctionReply
case object AuctionOver   extends AuctionReply
```

# Scala - Auction Example

```
class Auction(seller: Actor, minBid: Int, closing: Date) extends Actor {
  val timeToShutdown =    36000000 // inform that auction was closed
  val bidIncrement = 10
  def act() {
    var maxBid = minBid - bidIncrement;  var maxBidder: Actor = null; var running = true
    while (running) {
      receiveWithin ((closing.getTime() - new Date().getTime())) {
        case Offer(bid, client) =>
          if (bid >= maxBid + bidIncrement) {
            if (maxBid >= minBid) maxBidder ! BeatenOffer(bid)
            maxBid = bid; maxBidder = client; client ! BestOffer }
          else client ! BeatenOffer(maxBid)
        case Inquire(client) =>
          client ! Status(maxBid, closing)
        case TIMEOUT =>
          if (maxBid >= minBid) {
            val reply = AuctionConcluded(seller, maxBidder)
            maxBidder ! reply; seller ! reply }
          else seller ! AuctionFailed
          receiveWithin(timeToShutdown) {
            case Offer(_, client) =>
              client ! AuctionOver
            case TIMEOUT =>
              running = false }}}}}
```

# Scala - Concurrent Programming

- Alternative *react* function, also takes partial function as input for the decision, but does not return on match
  - Another tail recursion case - implementable by one thread
  - Message handler must process the message and do all remaining work
  - Typical idiom is to have top-level work method being called

```
object NameResolver extends Actor {
  import java.net.InetAddress
  def act() {
    react {
      case (name: String, actor: Actor) =>
        actor ! InetAddress.getByName(name)
        act()
      case „EXIT“ =>
        println(„Exiting“)
      case msg =>
        println(„Unknown message“)
        act()
}}
```

# Programming Models

<table>
<tr><th></th><th>Task-Parallel Programming Model</th><th>Data-Parallel Programming Model</th><th>Actor Programming Model</th><th>Functional Programming Model</th><th>PGAS / DSM Programming Model</th></tr>
<tr><td>Shared Memory System</td><td>OpenMP, Threading Libs, Linda, Ada, Cilk</td><td>OpenMP, PLINQ, HPF</td><td rowspan="2">Scala, Erlang</td><td>Lisp, Clojure, Haskell, Scala, Erlang</td><td>-</td></tr>
<tr><td>Distributed Memory System</td><td colspan="2">Socket communication, MPI, PVM, JXTA, MapReduce, CSP channels</td><td>Scala, Erlang</td><td>-</td></tr>
<tr><td>Hybrid System</td><td>-</td><td>OpenCL</td><td>-</td><td>-</td><td>Unified Parallel C, Titanium, Fortress, X10,Chapel</td></tr>
</table>