Dataflow, actors and high level structures in concurrent applications

Anthony Williams

Just Software Solutions Ltd
`http://www.justsoftwaresolutions.co.uk`

26th April 2012

Aims of High Level Approaches

Make it easier to write applications that ...

- Scale with hardware
- Are **obviously correct** rather than having **no obvious problems** — C.A.R. Hoare

Aspects of Application Design

4 Aspects:

- Tasks
- Communication
- State
- Concurrency

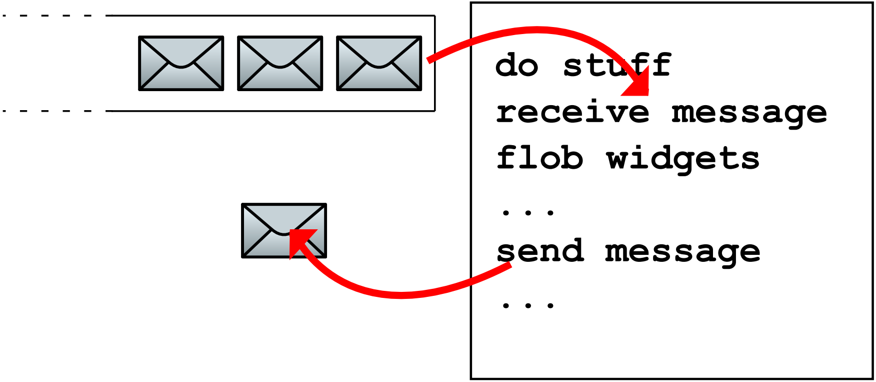
High Level Approaches

- Actors
- Active Objects
- Dataflow
- Loop Parallelism

High Level Approaches

- **Actors**
- Active Objects
- Dataflow
- Loop Parallelism

Actors



```
do stuff
receive message
flob widgets
...
send message
...
```

Actors in Erlang

- Process $\equiv$ Actor
- Messages are a language feature
- Guaranteed isolation

```
-export([ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

main(_) ->
    Pong_PID = spawn(?MODULE, pong, []),
    spawn(?MODULE, ping, [5, Pong_PID]),
    timer:sleep(1000).
```

```
-export([source/2, target/0]).

source(0, Target_PID) ->
    Target_PID ! finished,
    io:format("source finished~n", []);
source(N, Target_PID) ->
    io:format("source sending message ~w~n", [N]),
    Target_PID ! {message,N},
    source(N - 1, Target_PID).

dump_messages() ->
    receive
        {message,N} ->
            io:format("Target received message ~w~n", [N]),
            dump_messages()
    end.

target() ->
    receive
        finished ->
            io:format("Target finished~n", []),
            dump_messages()
    end.

main(_) ->
    Target_PID = spawn(?MODULE, target, []),
    spawn(?MODULE, source, [5, Target_PID]),
    timer:sleep(1000).
```

```erlang
target() ->
    receive
        finished ->
            io:format("Target finished~n", []),
            dump_messages();
        _ ->
            io:format("Unexpected message~n", []),
            target()
    end.
```

Dynamic actors

Actors can be started dynamically $\Rightarrow$ can add new actors in response to messages

```
chain_sieve(My_prime,Next_sieve) ->
    receive
        N -> if (N rem My_prime ) == 0 -> true;
                true ->
                     Next_sieve ! N
           end
    end,
    chain_sieve(My_prime,Next_sieve).

sieve(My_prime) ->
    io:format("~w~n",[My_prime]),
    receive
        N ->
            if (N rem My_prime ) == 0 ->
                    sieve(My_prime);
               true ->
                    Next_sieve = spawn(?MODULE,sieve,[N]),
                    chain_sieve(My_prime,Next_sieve)
            end
    end.
```

Actors in C++

- Actor ≈ Thread
- Actors are a library facility
- Isolation by programmer discipline

```
struct ping { jss::actor_ref sender; };
struct pong {};
struct finished {};

void pingfunc(unsigned n,jss::actor_ref pong_id){
  while(n--) {
    pong_id << ping{jss::actor::self()};
    jss::actor::receive().match<pong>(
      [](pong){
        std::cout<<"ping received pong\n";
      });
  }
  pong_id << finished();
  std::cout<<"ping finished\n";
}
```

```
void pongfunc() {
  bool done=false;
  while(!done) {
    jss::actor::receive()
      .match<ping>(
        [](ping p){
          std::cout<<"pong received ping\n";
          p.sender << pong();
        })
      .match<finished>(
        [&](finished){
          std::cout<<"pong finished\n";
          done=true;
        });
  }
}
```

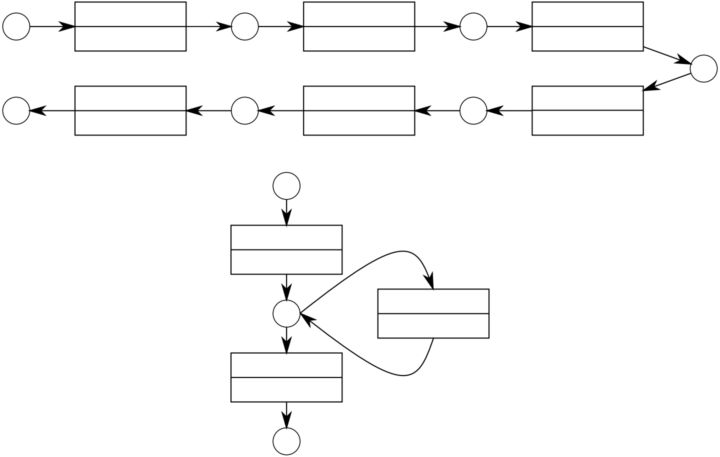
Actors in Scala

- Actors may share threads
- Actors are a library facility
- Isolation by programmer discipline

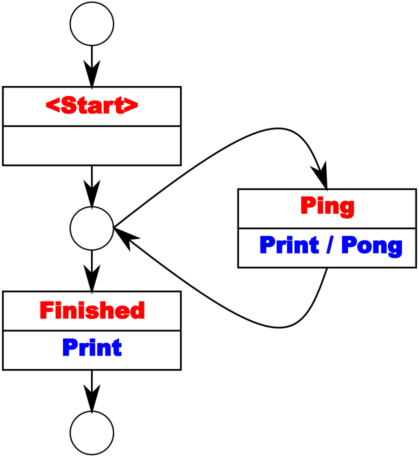
```
case object Ping
case object Pong
case object Finished

class Ping(count: Int, pong: Actor) extends Actor {
  def act() {
    var pingsLeft = count
    while(pingsLeft > 0) {
      pong ! Ping
      receive {
        case Pong =>
          Console.println("Ping received pong")
      }
      pingsLeft -= 1
    }
    Console.println("Ping finished")
    pong ! Finished
  }
}
```

```
class Pong extends Actor {
  def act() {
    loop {
      react {
        case Ping =>
          Console.println("Pong received ping ")
          sender ! Pong
        case Finished =>
          Console.println("Pong finished")
          exit()
      }
    }
  }
}
```

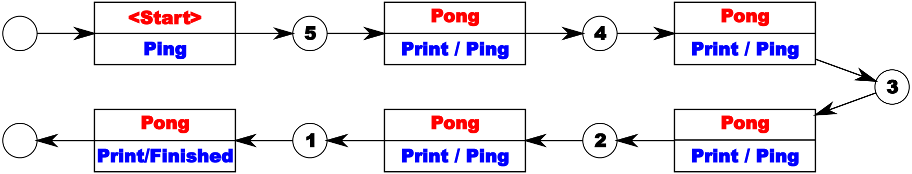
Actors as state machines



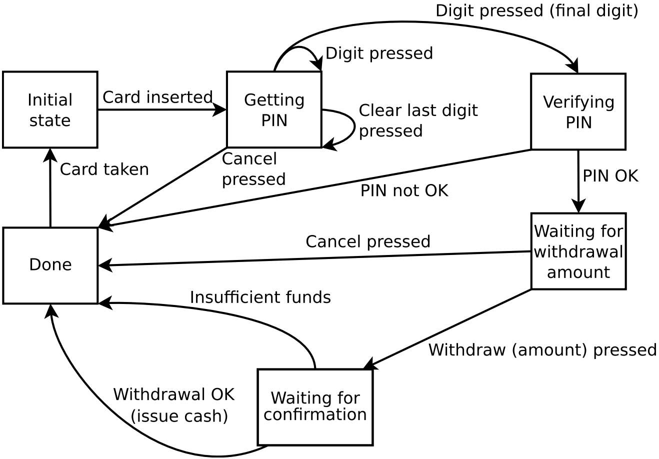
Actors as state machines (I)



Actors as state machines (II)



Actors as state machines (III)



```cpp
class atm {
  actor_ref bank;
  actor_ref interface_hardware;
  void (atm::*state)();

  std::string account;
  unsigned withdrawal_amount;
  std::string pin;
public:
  void operator()() {
    state=&atm::waiting_for_card;
    for(;;) {
      (this->*state)();
    }
  }
};
```

```
void wait_for_action() {
  interface_hardware<<display_withdrawal_options();
  actor::receive()
    .match<withdraw_pressed>(
      [&](withdraw_pressed const& msg) {
        withdrawal_amount=msg.amount;
        bank<<withdraw{account,msg.amount,actor::self()};
        state=&atm::process_withdrawal;
      })
    .match<balance_pressed>(
      [&](balance_pressed const& ) {
        bank<<get_balance{account,actor::self()};
        state=&atm::process_balance;
      })
    .match<cancel_pressed>(
      [&](cancel_pressed const& ) {
        state=&atm::done_processing;
      });
}
```

Actors: Summary

Tasks	Master function, message handlers
Communication	Message queues
State	Actor's internal state
Concurrency	Limited to number of actors

High Level Approaches

- Actors
- **Active Objects**
- Dataflow
- Loop Parallelism

Active Objects

- Special sort of actor
- Send messages by method calls
- Results returned in a future

Active Objects in Groovy

- Annotate the class with `@ActiveObject`
- Annotate the method with `@ActiveMethod`
- The return type is `DataflowVariable`

```
@ActiveObject
class DeepThought {
    @ActiveMethod
    def findTheAnswerToLifeTheUniverseAndEverything() {
        println "Thinking"
        sleep 5000
        println "Answer Ready"
        return 42
    }
}

final DeepThought dt=new DeepThought()
def theAnswer=dt.findTheAnswerToLifeTheUniverseAndEverything()
println "Doing stuff"
sleep 2000
println "Waiting"
println "The answer is ${theAnswer.get()}"
```

Active Objects in C++

- Do it manually with an actor
- Explicitly declare the return type as a future

```
struct find_the_answer{std::promise<int> promise;};
static void actor_loop() {
  for(;;){
    jss::actor::receive().match<find_the_answer>(
        [](find_the_answer fta) {
          std::cout<<"Thinking\n";
          std::this_thread::sleep_for(
            std::chrono::seconds(5));
          std::cout<<"Answer ready\n";
          fta.promise.set_value(42);
        });
  }
}
std::future<int> findTheAnswerToLifeTheUniverseAndEverything()
{
  find_the_answer fta;
  std::future<int> res=fta.promise.get_future();
  internal_actor<<std::move(fta);
  return res;
}
```

```
int main(){
  DeepThought dt;
  auto answer=dt.findTheAnswerToLifeTheUniverseAndEverything();
  std::cout<<"Doing stuff\n";
  std::this_thread::sleep_for(std::chrono::seconds(2));
  std::cout<<"Waiting\n";
  answer.wait();
  std::cout<<"The answer is "<<answer.get()<<std::endl;
}
```

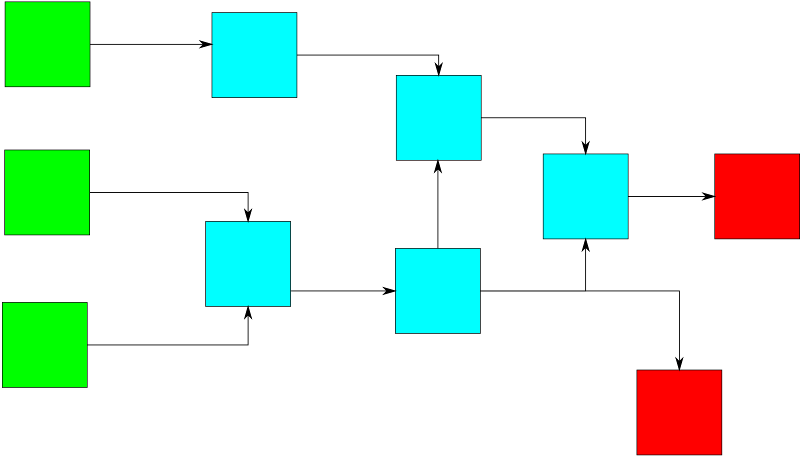
Active Objects: Summary

Tasks	Active methods
Communication	Method calls, futures
State	Active Object's internal state
Concurrency	Limited to number of Active Objects

High Level Approaches

- Actors
- Active Objects
- **Dataflow**
- Loop Parallelism

Dataflow Architectures (I)



Dataflow Architectures (II)

- Primary concern is the **flow** of data between tasks
- Tasks may be 1-1, 1-Many, Many-1 or Many-Many
- Tasks may have state

Dataflow Architectures (III)

Basic task types include:

- Generators
- Filters
- Routing operations
- Transforms

Dataflow Architectures (IV)

May define flows for:

- 1 set of inputs ⇒ 1 set of outputs
- A series of sets of inputs ⇒ a series of sets of outputs

Dataflow Variables

- Write-once
- May be assigned a value explicitly
- Value may be computed by a task

Dataflow variables in Groovy

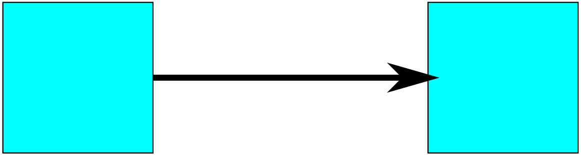
```
import groovyx.gpars.dataflow.DataflowVariable
import static groovyx.gpars.dataflow.Dataflow.task

final def a=new DataflowVariable()
final def b=task{
    return a.val + 10
}

a<<5;

println "Result: ${b.val}"
```

Dataflow variables in C++

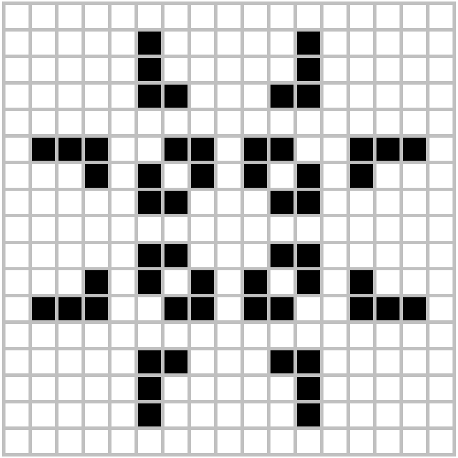
```
#include <jss/dataflow.hpp>
#include <iostream>

jss::dataflow::variable<int> a;
jss::dataflow::variable<int> b;

int main(){
    b.task([]{
            return a.get()+10;
        });
    a=5;
    std::cout<<"Result: "<<b.get()<<std::endl;
}
```

Dataflow channels

A channel ties tasks together



Conway's Game of Life



```
bool cell_rules(std::vector<bool> const& incoming){
  bool const was_alive=incoming[0];
  unsigned const alive_neighbours=
    std::count(incoming.begin()+1,incoming.end()-1,true);
  return (was_alive && (alive_neighbours==2)) ||
    (alive_neighbours==3);
}

void bind_cell_evolution_rules(){
  for(unsigned x=0;x<width;++x){
    for(unsigned y=0;y<height;++y){
      std::vector<jss::dataflow::readable_channel<bool> > vec=
        find_neighbours(x,y);
      vec.push_back(heartbeat);
      jss::dataflow::combine(vec).
        transform(cell_rules).write_to(cells[x][y]);
    }
  }
}
```

Dataflow: Summary

Tasks	Transforms, generators, etc.
Communication	Channels
State	Task's internal state
Concurrency	Items x tasks

High Level Approaches

- Actors
- Active Objects
- Dataflow
- **Loop Parallelism**

Loop Parallelism

- Declarative: do **this** for each of **these** data items
- Used in OpenMP, TBB, C++AMP

Loop Parallelism (II)

Parallel versions of:

- `std::for_each`
- `std::find`
- `std::count`
- `std::transform`
- `std::accumulate`

OpenMP naive matrix multiplication

```
#pragma omp parallel for
for (i = 0; i < nrows; i++){
  for(j = 0; j < ncols; j++){
    for (k = 0; k < nrowcols; k++){
       c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

This only parallelizes the outer loop

TBB naive matrix multiplication

```
parallel_for(
  blocked_range<int>(0,nrows),
  [&](blocked_range<int> r) {
    for (int i=r.begin();i!=r.end();++i) {
      parallel_for(
        blocked_range<int>(0,ncols),
        [&](blocked_range<int> r2) {
          for(int j=r2.begin();j!=r2.end();++j){
            for(int k=0;k<nrowcols;++k)
              c[i][j] += a[i][k] * b[k][j];
          }
        });
    }
  });
```

C++AMP matrix multiplication

```
concurrency::array_view<const float,2> va(
  nrows, nrowcols, a);
concurrency::array_view<const float,2> vb(
  nrowcols, ncols, b);
concurrency::array_view<float,2> vc(
  nrows, ncols, c); vc.discard_data();
concurrency::parallel_for_each(vc.extent,
[=](concurrency::index<2> idx) restrict(amp) {
  int row = idx[0]; int col = idx[1];
  float sum = 0.0f;
  for(int i = 0; i < W; i++)
    sum += va(row, i) * vb(i, col);
  vc[idx] = sum;
});
```

Loop Parallelism: Summary

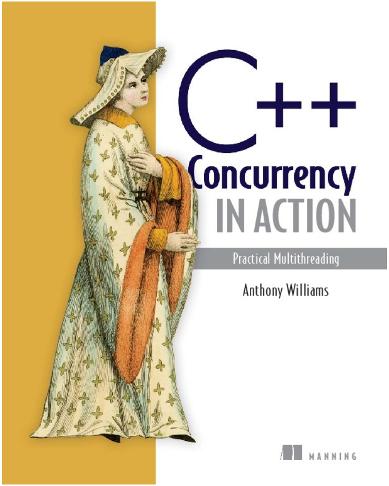
Tasks	Core loop function
Communication	Shared data
State	Shared data
Concurrency	Limited to number of data items

Just::Thread

`just::thread` provides a complete implementation of the C++11 thread library for MSVC and g++ on Windows, and g++ for Linux and MacOSX.

`Just::Thread` **Pro** also coming soon, with support for many of the high level facilities shown in this presentation. Find out more at:
`http://www.stdthread.co.uk/pro`

My Book



C++ Concurrency in Action: Practical Multithreading with the new C++ Standard.

`http://stdthread.com/book`