PCPP: PRACTICAL CONCURRENT & PARALLEL PROGRAMMING

MESSAGE PASSING CONCURRENCY II / II

Claus Brabrand
(((( brabrand@itu.dk ))))
Associate Professor, Ph.D.
(((( Software and Systems ))))
IT University of Copenhagen
AGENDA

3) Broadcast:
- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one ⇒ one-to-many)

AKKA: A proper introduction
- Motivations and benefits of Actors & Message Passing
- Recommendations

4) Primer:
- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

★ Scatter-Gather:
- Prototypical AKKA Service (dynamic load balancing)
- Extensions...
-module(helloworld).
-export([[start/0, account/1,bank/0,clerk/0]]).

%% -- BASIC PROCESSING -------------------
n2s(N) -> lists:flatten io_lib:format("~p", [N]). %% HACK!
random(N) -> random:uniform(N) div 10.

%% -- ACTORS -----------------------------
account(Balance) ->
    receive
      {deposit,Amount} ->
        account(Balance+Amount);
      {printbalance} ->
        io:fwrite(n2s(Balance) ++ "\n")
    end.

bank() ->
    receive
      {transfer,Amount,From,To} ->
        From ! {deposit,-Amount},
        To ! {deposit,+Amount},
        bank()
    end.

ntransfers(0,_,_,_) -> true;
ntransfers(N,Bank,From,To) ->
    R = random(100),
    Bank ! {transfer,R,From,To},
    ntransfers(N-1,Bank,From,To).

clerk() ->
    receive
      {start,Bank,From,To} ->
        random:seed(now()),
        ntransfers(100,Bank,From,To),
        clerk()
    end.

start() ->
    A1 = spawn(helloworld,account,[0]),
    A2 = spawn(helloworld,account,[0]),
    B1 = spawn(helloworld,bank,[
    ],
    B2 = spawn(helloworld,bank,[
    ],
    C1 = spawn(helloworld,clerk,[
    ],
    C2 = spawn(helloworld,clerk,[
    ],
    C1 ! {start,B1,A1,A2},
    C2 ! {start,B2,A2,A1},

MANDATORY HAND-IN!

a) Color ABC.erl
(according to color convention):
send, receive, msgs
actors, spawn, rest.
(try 2 B as consistent as possible)

MOTIVATION:

1) Structure of ERLANG:
- Syntax (structure); then
- Semantics (meaning)

2) Discern linguistic aspects:
send, receive, msgs
actors, spawn, rest.
PROBLEMS: Sharing & Mutability!

Thread T1

- OK (unshared)
- OK (immutable)
- DANGER (shared mutable)

Thread T2

- OK (unshared)

SOLUTIONS:
1) atomic access!
   locking or transactions
   (NB: avoid deadlock!)
2) avoid mutability!
3) avoid sharing...

Atomic access:
- Locking (pessimistic)
- Transactions (optimistic)
**PROBLEMS:**
Sharing & Mutability!

**SOLUTIONS:**
1) atomic access! locking or transactions (NB: avoid deadlock!)
2) avoid mutability!
3) avoid sharing...

Thread T1
---
 immutable datastructures

Thread T2
---
 No sharing!

Atomic access:
Locking (pessimistic) or Transactions (optimistic)

OK (unshared)

message

OK (unshared)
Philosophy & Expectations!

- **ERLANG:**
  - We'll use as message passing *specification language*
  - You have to-be-able-to *read* simple ERLANG programs
    - (i.e., not *write*, nor *modify*)

- **JAVA+AKKA:**
  - We'll use as msg passing *implementation language*
  - You have 2-b-a-2 *read/write/modify* JAVA+AKKA p's
  - However, we'll use its "*pure msg pass core*" only!

**NB:** we're *not* going to use all of its fantazilions of functions!
3) Broadcast

- **start**
- **broadcaster**
- **p1**
- **p2**
- **p3**

```plaintext
{subscribe, P1}
{subscribe, P2}
{subscribe, P3}
{message, "foo"}
{unsubscribe, P2}
{message, "bar"}
```

Messages:
- **foo!**
- **bar!**
3) **Broadcast.erl**

```erlang
-module(helloworld).
-export([start/0,person/0,broadcaster/1]).

person() ->
    receive
        {message,M} ->
            io:fwrite(M ++ "\n"),
            person()
    end.

broadcast([],_) -> true;
broadcast([Pid|L],M) ->
    Pid ! {message,M},
    broadcast(L,M).

broadcaster(L) ->
    receive
        {subscribe,Pid} ->
            broadcaster([Pid|L]) ;
        {unsubscribe,Pid} ->
            broadcaster(lists:delete(Pid,L)) ;
        {message,M} ->
            broadcast(L,M),
            broadcaster(L)
    end.
```

**start() ->**
- `Broadcaster = spawn(helloworld,broadcaster,[]),`
- `P1 = spawn(helloworld,person,[]),`
- `P2 = spawn(helloworld,person,[]),`
- `P3 = spawn(helloworld,person,[]),`
- `Broadcaster ! {subscribe,P1},`
- `Broadcaster ! {subscribe,P2},`
- `Broadcaster ! {subscribe,P3},`
- `Broadcaster ! {message,"Purses half price!"},`
- `Broadcaster ! {unsubscribe,P2},`
- `Broadcaster ! {message,"Shoes half price!!"}.`

**broadcast()**
- purses half price!
- purses half price!
- purses half price!
- shoes half price!!
- shoes half price!!
import java.util.*;
import java.io.*;
import akka.actor.*;

// -- MESSAGES

class SubscribeMessage implements Serializable {
    public final ActorRef subscriber;
    public SubscribeMessage(ActorRef subscriber) {
        this.subscriber = subscriber;
    }
}

class UnsubscribeMessage implements Serializable {
    public final ActorRef unsubscriber;
    public UnsubscribeMessage(ActorRef unsubscriber) {
        this.unsubscriber = unsubscriber;
    }
}

class Message implements Serializable {
    public final String s;
    public Message(String s) {
        this.s = s;
    }
}
3) Broadcast.java

```java
public class Broadcast {
    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("BroadcastSystem");
        final ActorRef broadcaster = system.actorOf(Props.create(BroadcastActor.class), "broadcaster");
        final ActorRef p1 = system.actorOf(Props.create(PersonActor.class), "p1");
        final ActorRef p2 = system.actorOf(Props.create(PersonActor.class), "p2");
        final ActorRef p3 = system.actorOf(Props.create(PersonActor.class), "p3");
        broadcaster.tell(new SubscribeMessage(p1), ActorRef.noSender());
        broadcaster.tell(new SubscribeMessage(p2), ActorRef.noSender());
        broadcaster.tell(new SubscribeMessage(p3), ActorRef.noSender());
        broadcaster.tell(new Message("purses half price!"), ActorRef.noSender());
        broadcaster.tell(new Message("shoes half price!!"), ActorRef.noSender());
        try {
            System.out.println("Press return to terminate...");
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}
```
3) Broadcast.java

// -- ACTORS -----------------------------------------------

class PersonActor extends UntypedActor {
    public void onReceive(Object o) throws Exc' {
        if (o instanceof Message) {
            System.out.println(((Message) o).s);
        }
    }
}

class BroadcastActor extends UntypedActor {
    private List<ActorRef> list =
        new ArrayList<ActorRef>();

    public void onReceive(Object o) throws Exception {
        if (o instanceof SubscribeMessage) {
            list.add(((SubscribeMessage) o).subscriber);
        } else if (o instanceof UnsubscribeMessage) {
            list.remove(((UnsubscribeMessage) o).unsubscriber);
        } else if (o instanceof Message) {
            for (ActorRef person : list) {
                person.tell(o, getSelf());
            }
        }
    }
}

person() ->
    receive
        {message,M} ->
            io:fwrite(M ++ "\n"),
            person()
            end.

broadcast([[],_] -> true;
broadcast([Pid|L],M) ->
    Pid ! {message,M},
    broadcast(L,M).

broadcaster(L) ->
    receive
        {subscribe,Pid} ->
            broadcaster([Pid|L]) ;
        {unsubscribe,Pid} ->
            broadcaster( L \ Pid ) ;
        {message,M} ->
            broadcast(L,M),
            broadcaster(L)
        end.
3) **Broadcast.java**

- **Compile:**
  
  ```sh
  javac -cp scala.jar:akka-actor.jar Broadcast.java
  ```

- **Run:**
  
  ```sh
  java -cp scala.jar:akka-actor.jar:akka-config.jar:. Broadcast
  ```

- **Output:**
  
  ```
  purses half price!
purses half price!
purses half price!
shoes half price!!
shoes half price!!
  ```
AGENDA

3) Broadcast:
- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one ⇒ one-to-many)

AKKA: A proper introduction
- Motivations and benefits of Actors & Message Passing
- Recommendations

4) Primer:
- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

★ Scatter-Gather:
- Prototypical AKKA Service (dynamic load balancing)
- Extensions
Mountain in Sweden:
- Northern Sweden
- (Close to Norway)

Nordic Goddesses: "Àhkkas"
- From Nordic/Arctic/Saami Mythology
- The Àhkkas: daughters of Mother Sun
- Ancient creator goddesses of the past

Software runtime middleware:
- For Java Virtual Machine (made in Scala)
Why the Sudden Popularity?!

Recently, processor speed "hit the wall":

- Speed of light:  \( c = 3 \cdot 10^8 \text{ m/s} \) (meters per sec)
- Processor speed:  \( s = 3 \cdot 10^9 \text{ x/s} \) (instructions per sec)
  - \( \frac{c}{s} = \frac{3 \cdot 10^8 \text{ m/s}}{3 \cdot 10^9 \text{ x/s}} = 0.1 \text{ m/x} \) (meters per instruction)
  - i.e., "light travels 10cm per instruction"

Before (more speed):

- Moore's Law: \#Transistors-per-CPU doubles every 1.8 years
- Dennard Scaling: Performance-per-Watt doubled every 1.57 years

Now (more speed):

- We have to increase parallelism!
- Recent developments: "Work Stealing Queue"

Chase and Lev: "Dynamic Circular Work-Stealing Queue", SPAA 2005
Michael, Vechev, Saraswat: "Idempotent Work Stealing", PPoPP 2009
Scalability? :-(

Using a conventional "Thread Pool Executor":

[ From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark ]
Scalability? :-(

- Using a conventional "Thread Pool Executor":

![Graph showing Throughput (msg/s) vs. number of actors]

- Bottleneck: Thread Queue!

[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]
Scalability!

- Java 8's New Fork Join Pool (Work Stealing):

Throughput (msg/s) vs. number of actors

[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]
Scalability!

- Java 8’s New Fork Join Pool (Work Stealing):

  - Previous graph: 20M msgs/sec
  - Previous graph: 1.5M msgs/sec

[ From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark ]
Scalability! :-)

...and after optimizing for throughput:

[ From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark ]
Scalability! :-)

...and after optimizing for throughput:

50+M msgs/sec

[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]
An Actor is...:

- A fundamental *unit of computation* that embodies:
  - 1) Processing
  - 2) State
  - 3) Communication

- In particular: no shared & mutable state!

- When an actor receives a message, it can...:
  - 1) perform computation
  - 2) change state
  - 3) send messages to actors it knows
  - 4) spawn new actors
Actors & Msg Passing: Benefits

- **Correct highly concurrent systems:**
  - Higher level abstraction (via message passing)
    - coordination (declarative/what) ≠ business logic (imperative/how)
  - No low-level locking (no shared && mutable state)

- ** Truly scalable systems:**
  - Actors are extremely lightweight entities
  - Actors are location transparent
  - Distributable-by-design
  - Transparently map MP programs onto given hardware:
    - "Scale up" (more processors), "Scale out" (more machines)

- **Self-healing, fault-tolerant systems:**
  - Adaptive load balancing and actor migration
  - "Let it crash" model (deal w/ failure, great succes in telecom industry)
  - Manage system overload (graceful service degradation)
Actors & Msg Passing: Drawbacks

- **New/different paradigm:**
  - Many programmers unfamiliar with message passing

- **Overhead of (high level) message passing:**
  - Overhead of sending messages:
    - Less efficient than shared & mutable state
  - (analogy: expl memory allocation vs garbage collection)

- **The Correlation problem:** [exemplified...]:
  - A --favorite-fruit?--> B ; A --favorite-color?--> B;
  - B --orange!--> A (correlate response with request?!?)

- **Hard to identify global state properties:**
  - i.e., computing: \( f(\text{global-state}) \):
    - e.g., termination condition of a distributed algorithm
The People Metaphor

- **Shared Mutable state:**
  - Concurrency problems!
  - Shared && mutable!
  - Hard to (later) distribute!

- **Actors (with encapsulated state):**
  - Can't "look inside head" of a living person (actor)!
  - Instead: ask questions?
  - ...and get responses!
  - ...one at a time!

actor is only doing one thing at a time
The People Metaphor (cont'd)

- Programming Metaphor:
  - "The People Analogy"

- Think of it as "coordinating lots of people":
  - Each can do simple tasks
  - Consider workflow (of orders/messages in your system)
The People Metaphor (cont'd)

- Programming Metaphor:
  - "The People Analogy"

- Think of it as "coordinating lots of people":
  - Each can do simple tasks
  - Consider workflow (of orders/messages in your system)
  - Need more work ⇒ hire more people (spawn more actors)
The People Metaphor

Programming Metaphor:
- "The People Analogy"

Think of it as "coordinating lots of people":
- Each can do simple tasks
- Consider workflow (of orders/messages in your system)
- Need more work ⇒ hire more people (spawn more actors)
- Hierarchic organization: managers supervise workers
- Fault tolerance (expect failures and deal with them)
Recommendations

1) Actors should be like nice co-workers:
   - do their job effectively w/o bothering others needlessly
   - should not roll thumbs (idle or blocking operations)

2) Actors should not send mutable objects:
   - O/w we're back to "shared && mutable" ⇒ problems!

3) Actors should send data, not programs:
   - O/w we're back to "shared && mutable" ⇒ problems!
   - (Note: ERLANG does not have higher-order functions)

4) Create few top-level actors:
   - If these crash, your whole system will crash
   - If their workers die, they just hire (spawn) new ones
Recommendations

5) Managers should supervise Workers:
   - organization as a hierarchy
   - pass on and schedule tasks for workers
   - "hire" (spawn) more workers by need
   - deal with failure (of your workers)

6) Actors should spawn workers for "dangerous operations" (Qatar 2022 ?):
   - avoid crashing with important data
   - spawn workers for "dangerous operations"
   - deal with failure (of your workers)
A Hierarchy of Actors

[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]
[ From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark ]
Scale Down, Scale Up, Scale Out

When programming with actors and...:

- Scale Up
- Scale Out
- Scale Down

(message passing)

(more processors)

(more machines)
Fault Tolerance

[ From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark ]
[ From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark ]
AGENDA

3) Broadcast:
- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one ⇒ one-to-many)

AKKA: A proper introduction
- Motivations and benefits of Actors & Message Passing
- Recommendations

4) Primer:
- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

★ Scatter-Gather:
- Prototypical AKKA Service (dynamic load balancing)
- Extensions
4) Primer

Let's assume...:
FAST message passing, SLOW i/o and CPU
Q: Which output order ?

Let's assume...:
SLOW message passing, FAST i/o and CPU
Q: Which output order ?

{init,N} ⇒ create N slave workers

{isprime, P} ⇒ delegate to P%N :-)

Claus Brabrand, ITU, Denmark

PRACTICAL CONCURRENT AND PARALLEL PROGRAMMING (PCPP)
4) Primer.erl

-module(helloworld).
-export([start/0, slave/1, primer/1]).

is_prime_loop(N,K) ->
    K2 = K * K, R = N rem K,
    case (K2 =< N) and (R /= 0) of
        true  -> is_prime_loop(N, K+1);
        false -> K
    end.

is_prime(N) ->
    K = is_prime_loop(N,2),
    (N >= 2) and (K*K > N).

n2s(N) ->
    lists:flatten(io_lib:format("~p",[N])).

slave(Id) ->
    receive
        {isprime, N} ->
            case is_prime(N) of
                true -> io:fwrite("(" ++
                          n2s(Id) ++ ") " ++ n2s(N) ++ "\n");
                false -> []
            end,
            slave(Id)
        end.
end.

create_slaves(Max,Max) -> [];
create_slaves(Id,Max) ->
    Slave = spawn(helloworld, slave, [Id]),
    [Slave|create_slaves(Id+1,Max)].

primer(Slaves) ->
    receive
        {init, N} when N=<0 ->
            throw({nonpositive,N}) ;
        {init, N} ->
            primer(create_slaves(0,N));
        {isprime, _} when Slaves == [] ->
            throw({uninitialized}) ;
        {isprime, N} when N=<0 ->
            throw({nonpositive,N}) ;
        {isprime, N} ->
            SlaveId = N rem length(Slaves),
            lists:nth(SlaveId+1, Slaves) ! {isprime,N},
            primer(Slaves)
    end.

spam(_, N, Max) when N>=Max -> true;
spam(Primer, N, Max) ->
    Primer ! {isprime, N},
    spam(Primer, N+1, Max).

start() ->
    Primer = 
        spawn(helloworld, primer, [[]]),
    Primer ! {init,7},
    spam(Primer, 2, 100).
import java.util.*;
import java.io.*;
import akka.actor.*;

// -- MESSAGES --------------------------------------------------
class InitializeMessage implements Serializable {
    public final int number_of_slaves;
    public InitializeMessage(int number_of_slaves) {
        this.number_of_slaves = number_of_slaves;
    }
}

class IsPrimeMessage implements Serializable {
    public final int number;
    public IsPrimeMessage(int number) {
        this.number = number;
    }
}
4) Primer.java

```java
// -- SLAVE ACTOR -------------------------------

class SlaveActor extends UntypedActor {
    private boolean isPrime(int n) {
        int k = 2;
        while (k * k <= n && n % k != 0) k++;
        return n >= 2 && k * k > n;
    }

    public void onReceive(Object o) throws Exception {
        if (o instanceof IsPrimeMessage) {
            int p = ((IsPrimeMessage) o).number;
            if (isPrime(p)) System.out.println("(" + p % Primer.P + ") " + p); // HACK
        }
    }
}
```
Primer.java

```
// -- PRIME ACTOR --------------------------------------------------
class PrimeActor extends UntypedActor {
    List<ActorRef> slaves;

    private List<ActorRef> createSlaves(int n) {
        List<ActorRef> slaves = new ArrayList<ActorRef>();
        for (int i=0; i<n; i++) {
            ActorRef slave =
                getContext().actorOf(Props.create(SlaveActor.class), "p" + i);
            slaves.add(slave);
        }
        return slaves;
    }

    public void onReceive(Object o) throws Exception {
        if (o instanceof InitializeMessage) {
            InitializeMessage init = (InitializeMessage) o;
            int n = init.number_of_slaves;
            if (n<=0) throw new RuntimeException("*** non-positive number!");
            slaves = createSlaves(n);
            System.out.println("initialized (" + n + " slaves ready to work)!");
        } else if (o instanceof IsPrimeMessage) {
            if (slaves==null) throw new RuntimeException("*** uninitialized!");
            int n = ((IsPrimeMessage) o).number;
            if (n<=0) throw new RuntimeException("*** non-positive number!");
            int slave_id = n % slaves.size();
            slaves.get(slave_id).tell(o, getSelf());
        }
    }
}
```
public class Primer {
    private static void spam(ActorRef primer, int min, int max) {
        for (int i=min; i<max; i++) {
            primer.tell(new IsPrimeMessage(i), ActorRef.noSender());
        }
    }

    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("PrimerSystem");
        final ActorRef primer =
                system.actorOf(Props.create(PrimeActor.class), "primer");
        primer.tell(new InitializeMessage(7), ActorRef.noSender());
        try {
            System.out.println("Press return to initiate...");
            System.in.read();
            spam(primer, 2, 100);
            System.out.println("Press return to terminate...");
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            System.out.println("Press return to terminate...");
            System.in.read();
        }
    }
}
4) **Primer.java**

- **Compile:**
  
  ```
  javac -cp scala.jar:akka-actor.jar Primer.java
  ```

- **Run:**
  
  ```
  java -cp scala.jar:akka-actor.jar:akka-config.jar:. Primer
  ```

- **Output:**
  
  ```
  press return to initiate...
  initialized (7 slaves ready to work)!
  (2) 2
  (3) 3
  Press return to terminate...
  (0) 7
  (5) 5
  (4) 11
  (6) 13
  (3) 17
  (5) 19
  (2) 23
  (1) 29
  (3) 31
  (2) 37
  (6) 41
  (1) 43
  (5) 47
  (4) 53
  (3) 59
  (5) 61
  (4) 67
  (1) 71
  (3) 73
  (2) 79
  (6) 83
  (5) 89
  (6) 97
  ```
**ERLANG -vs- JAVA+AKKA**

**ERLANG:**

SLOW message passing*

= predicted = effect =>
You would get numbers in
slave-worker order:

- 07, 29, 02, 03, 11, 05, ...

[observed in ERLANG]

**JAVA+AKKA:**

FAST message passing*

= predicted = effect =>
You would get numbers in
numerical order:

- 02, 03, 05, 07, 11, 13, ...

[observed in JAVA+AKKA]

*) relative to computation, I/O, ...
Low vs High Parallelization!

Note: added silly time-consuming computation to every `isPrime()` method call!

- MacBook Air 1.8 GHz i5
- MTLab Server 2.8 GHz AMD

2x2 cores = 4 parallel actors
2x16 cores = 32 parallel actors

12 sec → 7.5 sec (1.6x speed up)
22 sec → 3.3 sec (6.7x speed up)

#actors

with even more processors (e.g., remoting)?
Note: Two primers with $P=3 \times Q=5$ is equivalent to one primer with $P=15$.
BTW: This is why you should always use a prime number in a hash function!
AGENDA

3) Broadcast:
- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one ⇒ one-to-many)

AKKA: A proper introduction
- Motivations and benefits of Actors & Message Passing
- Recommendations

4) Primer:
- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

★ Scatter-Gather:
- Prototypical AKKA Service (dynamic load balancing)
- Extensions
6) Scatter-Gather
6) Scatter-Gather

![Scatter-Gather Diagram]

- **Start**
  - **Worker**
    - **Split**
  - **Scatter**
    - **Worker**
    - **Worker**

- **Start**
  - **Split**
  - **Spawn**
    - **Worker**
    - **Worker**
6) Scatter-Gather

Scatter:
I don't want the result, send it to my gatherer:
- I'm too busy processing new incoming requests
- besides, it's hard to correlate requests-responses (aka, "the correlation problem")

Gatherer:
collect incoming responses and merge to final result inform its parent

--- OUTPUT ---

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>result = 2.5</td>
<td></td>
</tr>
</tbody>
</table>
6) Scatter-Gather

![Diagram of Scatter-Gather process]

- **Start** node initiates the process.
- **Scatter** node distributes work to multiple **Worker** nodes.
- **Split** and **Spawn** operations are used to divide and initiate worker tasks.
6) Scatter-Gather

Start

scatter

Worker

scatter

Worker

scatter

Worker

scatter

Worker

Gather

3.25

spawn

Gather

3

spawn

Gather

3.5

spawn

Gather

3

-- OUTPUT --

result = 3.25
6) ScatterGather.erl

-module(helloworld).
-export([start/0, worker/0, scatter/2, gather/1]).

%% -- COMPUTE --------------------------------------------------

seed() -> {_, A2, A3} = now(), % Seed wrt Time & Pid!
    random:seed(erlang:phash(node(), 100000), erlang:phash(A2, A3), A3).

n2s(N) -> lists:flatten(io_lib:format("~p", [N])). %% HACK: num to string conversion!

random(N) -> random:uniform(N).

compute(X) -> random(X).

average(X,Y) -> (X + Y) / 2.

%% -- START --------------------------------------------------

start() ->
    Worker = spawn(helloworld, worker, []),
    Worker ! split,
    Worker ! split,
    Worker ! {compute, 6, self()},
    receive
        {result, R} ->
            io:fwrite("result = " ++ n2s(R) ++ "\n")
    end.

-- OUTPUT --

6
1
2
4
result = 3.25
% -- WORKER

worker() ->
    seed(),
    receive
        split ->
            Left = spawn(helloworld,worker,[]),
            Right = spawn(helloworld,worker,[]),
            scatter(Left, Right);
        {compute,X,Caller} ->
            Res = compute(X),
            io:fwrite(n2s(Res) ++ "\n"),
            Caller ! {result,Res},
            worker()
    end.

% -- SCATTER

scatter(Left, Right) ->
    receive
        split ->
            Left ! split,
            Right ! split;
        {compute,X,Caller} ->
            Gather = spawn(helloworld,gather,[Caller]),
            Left ! {compute,X,Gather},
            Right ! {compute,X,Gather}
    end,
    scatter(Left, Right).

% -- GATHER

gather(Caller) ->
    receive
        {result,Res1} ->
            receive
                {result,Res2} ->
                    Res = average(Res1,Res2),
                    Caller ! {result,Res} % die!
            end
    end
end.

-- OUTPUT --

6
1
2
4
result = 3.25
import java.util.Random; import java.io.*;
import akka.actor.*;

// -- MESSAGES --------------------------------------------------
class StartMessage implements Serializable { public StartMessage() { } }
class SplitMessage implements Serializable { public SplitMessage() { } }
class CallerMessage implements Serializable {
    public final ActorRef caller;
    public CallerMessage(ActorRef caller) { this.caller = caller; }
}
class ComputeMessage implements Serializable {
    public final int number;
    public final ActorRef caller;
    public ComputeMessage(int number, ActorRef caller) {
        this.number = number;
        this.caller = caller;
    }
}
class ResultMessage implements Serializable {
    public final double result;
    public ResultMessage(double result) { this.result = result; }
}
class WorkerScatterActor extends UntypedActor {

    // null => worker, non-null => scatter:
    private ActorRef left, right;
    private final Random rnd = new Random();
    private int random(int n) { return rnd.nextInt(n); }
    private int compute(int n) { return random(n) + 1; }

    private void worker(Object o) throws Exception {
        if (o instanceof SplitMessage) {
            left = getContext().actorOf(Props.create(WorkerScatterActor.class), "left");
            right = getContext().actorOf(Props.create(WorkerScatterActor.class), "right");
        } else if (o instanceof ComputeMessage) {
            ComputeMessage m = (ComputeMessage) o;
            int result = compute(m.number);
            System.out.println(result);
            m.caller.tell(new ResultMessage(result), ActorRef.noSender());
        }
    }

    private void scatter(Object o) throws Exception { [...] }

    public void onReceive(Object o) throws Exception {
        // dispatch according to actor role: 'worker' or 'scatter'
        if (left == null) worker(o);
        else scatter(o);
    }
}
6) ScatterGather.java

class WorkerScatterActor extends UntypedActor {
    // null => worker , non-null => scatter:
    private ActorRef left, right;
    [
    ...
]

    private void worker(Object o) throws Exception { [...] }

    private void scatter(Object o) throws Exception {
        if (o instanceof SplitMessage) {
            left.forward(o, getContext());
            right.forward(o, getContext());
        } else if (o instanceof ComputeMessage) {
            ComputeMessage m = (ComputeMessage) o;
            ActorRef gather = getContext().actorOf(Props.create(GatherActor.class), "g");
            // send message with caller, instead of arguments to gather constructor:
            gather.tell(new CallerMessage(m.caller), ActorRef.noSender());
            left.tell(new ComputeMessage(m.number, gather), ActorRef.noSender());
            right.tell(new ComputeMessage(m.number, gather), ActorRef.noSender());
        }
    }

    public void onReceive(Object o) throws Exception {
        // dispatch according to actor role: 'worker' or 'scatter'
        if (left == null) worker(o);
        else scatter(o);
    }
}

-- OUTPUT --

6
1
2
4
result = 3.25
class GatherActor extends UntypedActor {
    double res1;
    ActorRef caller;

    private double average(double x, double y) {
        return (x + y) / 2;
    }

    public void onReceive(Object o) throws Exception {
        if (o instanceof CallerMessage) {
            caller = ((CallerMessage) o).caller;
        } else if (o instanceof ResultMessage) {
            if (caller == null) throw new Exception("no caller address!!!");
            if (res1 == 0) {
                res1 = ((ResultMessage) o).result;
            } else {
                double res2 = ((ResultMessage) o).result;
                double res = average(res1, res2);
                caller.tell(new ResultMessage(res), ActorRef.noSender());
                getContext().stop(getSelf()); // die!
            }
        }
    }
}

-- OUTPUT --

6
1
2
4
result = 3.25
6) ScatterGather.java

```java
class StartActor extends UntypedActor {
  public void onReceive(Object o) throws Exception {
    if (o instanceof StartMessage) {
      ActorRef worker =
        getContext().actorOf(Props.create(WorkerScatterActor.class), "worker");
      worker.tell(new SplitMessage(), ActorRef.noSender());
      worker.tell(new SplitMessage(), ActorRef.noSender());
      worker.tell(new ComputeMessage(6, getSelf()), ActorRef.noSender());
    } else if (o instanceof ResultMessage) {
      double result = ((ResultMessage) o).result;
      System.out.println("result = " + result);
    }
  }
}

public class ScatterGather {
  public static void main(String[] args) {
    final ActorSystem system = ActorSystem.create("HelloWorldSystem");
    final ActorRef starter =
      system.actorOf(Props.create(StartActor.class), "starter");
    starter.tell(new StartMessage(), ActorRef.noSender());
    try { System.out.println("Press return to terminate..."); System.in.read();
    } catch(InterruptedException e) { e.printStackTrace();
    } finally { system.shutdown();
  }
}
```

--- OUTPUT ---

```
6
1
2
4

result = 3.25
```
Scatter-Gather + ...

- **Adaptive Load balancing:**
  - Monitor system to extract up-to-date statistics
  - Based on statistics, adjust system capacity (cf. our split) or Quality-of-Service (ak²a, "graceful degradation")
    - Note: this may be done on all nodes in the hierarchy!

- **Memoization/Caching:**
  - Often, memoization is used to "cache" already-performed-computations
    ```
    // Map<Key,Val> cache;
    ```
    - Note: this may be done on all nodes in the hierarchy!

- **Fault Tolerance:**
  - Supervisors react if workers don't respond or crash
  - Then: resume(), subtree.restart(), parent.escalate()
More information...

- **ERLANG:**

- **AKKA Video Talks:**
  - [https://www.youtube.com/watch?v=GBvtE61Wrto](https://www.youtube.com/watch?v=GBvtE61Wrto)
  - [https://www.youtube.com/watch?v=t4KxWDqGfcs](https://www.youtube.com/watch?v=t4KxWDqGfcs)

- **JAVA+AKKA Documentation:**
  - [http://doc.akka.io/docs/akka/2.3.7/AkkaJava.pdf](http://doc.akka.io/docs/akka/2.3.7/AkkaJava.pdf)

- **JAVA+AKKA API:**
  - [http://doc.akka.io/japi/akka/2.3.7/](http://doc.akka.io/japi/akka/2.3.7/)
Thx!

Questions?
Reception (ERLANG vs AKKA)

- In ERLANG:
  - Locally nested receives (depending on local state)

- In JAVA+AKKA:
  - You only have one top-level receive:

Example ⇒ refactored (ready) for JAVA+AKKA:

```erlang
%% -- GATHER -----------------------------
gather(Pid) ->
  receive // State #0 ('Res1' not set)
    {result,Res1} ->
      receive // State #1 ('Res1' set)
        {result,Res2} ->
          Res = average(Res1,Res2),
          Pid ! {result, Res} % die.
      end
  end.

%% -- GATHER' ----------------------------
gather(Pid, Res1) ->
  receive
    {result,Res1} when Res1 = undef ->
      gather(Pid, Res1)
  ;
    {result,Res2} ->
      Res = average(Res1, Res2),
      Pid ! {result, Res} % die.
  end.
```

[See also ERLANG Book, program 5.3]