Principles Of Programming languages

-- Parallel and Distributed Languages
Why parallel?

• Some problems are naturally parallel
  – Parallel language for easier implementation
  – Parallel language “paradigm”

• For gaining efficiency
  – Divide work over multiple processors

• Some problems require a parallel implementation
  – Require too much memory
  – Require too much processing power/time
Motivation

• Question:
  • Cars on highway with stoplights
  • How many stoplights, cars before a traffic jam occurs?

• Use a discrete event simulator
  – Clocks
    • All naturally independent
  – Random event generators
    • new/delete/speedup/speed-down car,
    • stoplight switch
    • All naturally independent
  – Entities
    • All naturally operate independently
Programming environment

- Virtual CPUs
  - Can be created on demand

- Communication substrate
  - Shared memory
    - One processor writes to memory, another reads
    - “multiprocessor”
  - Distributed memory
    - Send/receive messages over a network
    - “distributed system”
  - Different paradigms for programming
    Shared+Distributed substrates
Parallel Problems

- **Deadlock**: two (or more) CPUs symmetrically wait for a resource the other currently has.

- **Livelock**: Both actively, symmetrically wait for a single resource (backing off simultaneously on failure).

- **Load-balancing**: each processor should have equal amount of work.
  - tradeoff: optimal load-balancing against lowering communication needs.

- **Data-race**: a processor modifies a data-structure while another is reading it.
  - can see inconsistent intermediate state.
co-routines

- Pseudo parallel routine invocation
  - “Cooperative multi-threading”
  - Voluntarily give up control to some other process

```
procedure Odd();
var i: integer;
begin
  for i := 1 to N by 2 do
    begin
      print i * i;
      resume Even;
    end
end;

procedure Even();
var i: integer;
begin
  for i := 2 to N by 2 do
    begin
      print i * i;
      resume Odd;
    end
end;
```

Odd();
co-routines

- Grafted on top of an functional/imperative language

- Notes:
  - Each 'resume' requires that all local variables/parameters be saved
    - Also those of the caller(s)
  - Co-routines are only pseudo parallelism
    - Each runs mutually exclusive from the other(s)
parallel-statements

- Grafted on top of an functional/imperative language

```par
par_begin
  statement 1
  statement 2
  statement 3
par_end

par_begin
  begin
  statements1
  end
begin
  begin
    statements2
  end
end
par_end

parfor i := 1 to 10 do
  begin
    statement(i);
  end
```
A, B, C: array \([1 .. N]\) of real;

\[
\text{parfor } i := 1 \text{ to } N \text{ do }
\begin{align*}
\text{begin} \\
\quad C[i] := A[i] + B[i]; & \quad (* \text{ all run in parallel } *) \\
\text{end}
\end{align*}
\]

\[
; \quad (* \text{ wait here till all parallel blocks finish } *)
\]

*problem: very fine granular: cost of allocating a virtual processor!
-- matrix multiplication: A, B input matrix, C output matrix
A,B,C: array of [1..N, 1..N] of real;

parfor i := 1 to N do
  parfor j := 1 to N do
    -- mult A[i,*] and B[*j]
    C[i, j] := 0;
    for k := 1 to N do
      C[i,j] := C[i,j] + (A[i, k] + B[k, j]);
    done
  done
done

Problem: inner, outer or both as parfor / normal-for?
Conclusion: programmer needs to tune granularity (the amount of work performed in a parallel block)
Parallel statements

• Solves the *easy* problem of allocating virtual processors

• Does not:
  – Help structure parallelism
  – Ensure safe access to data shared between virtual processors
    • data-races

• Deadlock / live-lock
  – Not possible as they're too simple in this form
Processes

- very common paradigm
- fork-join is the same as invoking a procedure
  - Child and parent run in parallel
    - Child and parent have their own address spaces
    - How do parent and child 'see' modifications of the other?
  - How do they recombine?

```
-- parent calls 'fork' to start new child
-- which starts at processType
fork processType(parameters);
-- by parent to wait for exit of child
join;

process processType(parameters);
begin
  -- notifies parent of child exit
  exit;
end;
```
process compile(sourcefile, targetfile : string); -- child
begin
    compile source file;
    store generated object file in targetfile;
    exit; -- terminate sub process/thread
end;

process main(void); -- parent process
begin
    -- create two parallel processes/threads
    fork compute("file1.c", "file1.o");
    fork compute("file2.c", "file2.o");
    join;
end;
Processes

- Variables/objects to be shared need to be declared 'shared'
  - Multi-processor: allocate them in shared memory
  - Distributed memory: send messages for reads/writes
- What if one wants to write concurrent to another's read?
  - Language level constructs/guarantees?
Example: producer-consumer

• a/some process(es) produce data, other(s) consume

• Example:
  – Pipelining
    • Each stage in a pipe accepts input from previous stage and passes it to the next stage
    • While washing pants[i], we can dry pants[i-1], etc
    • We need to pass pants to next stage each time
    • Washer must wait for dryer to empty
    • Dryer must wait for washer to finish
Example: producer-consumer

```plaintext
count: integer := 0;
buffer: array [ 0 .. MAX-1 ] of cloth_type;

process Washer;
  writepos: integer := 0;
  line: cloth_type;
begin
  while ... do
    -- should get input from somewhere
    line = get_and_clean_pants;
    while count == MAX do done
    buffer[write_pos] := line;
    write_pos := (write_pos + 1) mod MAX;
    count := count + 1;
  done
end;

process dryer;
  read_pos: integer := 0;
  line: cloth_type;
begin
  while ... do
    while count = 0 do done;
    line = buffer[read_pos];
    read_pos := read_pos + 1;
    count := count – 1;
    dry(line); -- 'dry' should pass output
    -- to next stage
  done
end

*problem: count inc/dec concurrent - race-condition
*problem: busy-waiting wastes cpu power - busy-waiting / polling
```
No language-level statement is atomic!

```plaintext
count := count + 1
```

translates to:

```plaintext
tmp := memory_load(&count);
tmp := tmp + 1;
memory_store(&count, tmp);
```

Example:
2 processors increment the same variable.
Expect $5 + 1 + 1 = 7$
Get 6!!
No language level statement is atomic!

--- Some 64 bit RISC architectures translate:

```plaintext
var x: long := 0xaabaabbbccccdddd; -- 64 bit

-- to:

store_upper_upper_16(&x, 0xaaa);
store_lower_upper_16(&x, 0xbbbb);
store_upper_lower_16(&x, 0xcccc);
store_lower_lower_16(&a, 0xdddd);
```

--- processor 1: store 0xaaaabbbbccccdddd;
store_upper_upper_16(&x, 0xaaa); -- win
store_lower_upper_16(&x, 0xbbbb); -- win
store_upper_lower_16(&x, 0xcccc); -- lose
store_lower_lower_16(&a, 0xdddd); -- lose

--- processor 1: store 0xeeeeffff11112222
store_upper_upper_16(&x, 0xeee); -- lose
store_lower_upper_16(&x, 0xffff); -- lose
store_upper_lower_16(&x, 0x11111); -- win
store_lower_lower_16(&a, 0x2222); -- win

--- processor 1: store 0xaaaabbbb11112222

Memory: x = 0xaabaabbb11112222
Synchronization

• Mutual exclusion synchronization
  – Exclude others from accessing some data
  – “critical section”
    • semantics: you are not excluding others from some piece of code, you protect data

• Condition synchronization
  – Wait for some condition to become true
Mutual-exclusion-sync.: locks

- lock(L);
  - Atomically/indivisibly acquires/sets lock, no other thread/process can acquire until this process/thread unlocks

- unlock(L);
  - Unlocks so that another process/thread can acquire the lock

SharedLock: Lock;

Washer -->
lock(SharedLock);
count := count + 1;
unlock(SharedLock);

Dryer -->
lock(SharedLock);
count := count - 1;
unlock(SharedLock);
Mutual-exclusion-sync.: semaphores

- **P(s)**
  - Increment 's' by one
  - Similar to unlock

- **V(s)**
  - If s = 0 then *wait for s > 0*
  - Decrement 's' by one
  - Similar to lock

- Slightly more generic than lock/unlock
Mutual-exclusion-sync.: semaphores

-- Add to washer-dryer example:

washerworking : semaphore := 0;
dryerempty : semaphore := 1;

-- add to washer

while ..
  V(dryerempty); -- wait for dryer
  -- hand over washing
  P(washerworking); -- tell dryer we're working

-- add to dryer

while ..
  V(washerworking); -- wait for washer
  -- get washing from washer
  P(dryerempty); -- tell washer were open for business
Mutual-exclusion-sync

- Problems
  - Hard to say which lock protects which variables
  - Locking granularity
    - Protect complete data-structure
      - Easier to program
    - Protect part of a data-structure
      - More efficient? (most of the time yes)
      - Cost of semaphore/lock/unlock is non-trivial
  - Locking strategy
    - Reads are safe, writes require locking?
      - Copy on write?
    - Update others
    - Optimistic vs pessimistic locking strategy
    - See 'parallel algorithms lecture', out-of-scope
Mutual exclusion-sync: monitors

- Special type of abstract-data-type
  - Access to data is via operations only
  - Operations are executed mutually exclusive to each other
    - Via internal/invisible semaphores/locks
  - Condition variables
    - Wait blocks until signaled
    - Signal wakes up one waiter
- Language level entity
  - locks/semaphore are mostly library things
    - Most languages however guarantee no code movement over locks/semaphores
monitor buffer;
   -- shared variables
   count: Integer := 0;
   buffer: array[0..MAX-1] of line;
   write_pos, read_pos: integer := 0;
   NotFull, NotEmpty: ConditionVariable;

operation put(in line: line_type);
begin
   if count = MAX then wait(NotFull); end if;
   buffer[write_pos] := line;
   write_pos := (write_pos + 1) mod MAX;
   count := count + 1;
   if count = 1 then signal(NotEmpty) end if;
end

operation get(in line: line_type);
begin
   if count = 0 then wait(NotEmpty); end if;
   line := buffer[read_pos];
   read_pos := (read_pos + 1) mod MAX;
   count := count – 1;
   if (count = MAX-1) then signal(NotFull) end if;
end

*Example languages using monitors: Mesa and Concurrent Pascal*
Message passing

- Distributed computers communicate over a network with explicit message sends
  - Each machine has its own memory
  - Each machine has its own clock

- Basic idea
  - Send M to R
  - Receive M' from R
    - M' a copy of M

- Effectively a distributed assignment statement
  - A reference to a copy of M is assigned to M'
Message passing: language level

- Communicating Sequential Processes (CSP)
  - Used in the occam language

- In practice: what data types are allowed in a message
  - Messages are typed
    - Arrays, primitives, records, not (in general) pointers, trees, lists
  - Send and receive are blocking

```plaintext
process Compiler;
begin
    while .. do
        line := generate line of assembly
        add_line_to_message(line);
        send message to Assembler
    end;
end;

process Assembler;
begin
    while .. do
        -- wait for slow compiler
        receive M from Compiler;
        line := extract_line(M);
        assembler_to_object_code(line);
    end;
end;
```
Example: occam

- Occam has
  - Parallel blocks and sequential blocks
    - Need to explicitly specify which
    - Parallel processes are started from parallel blocks
  - input/output streams for device/process communication
    - If X is a channel (notation: CHAN <type>)
      - Then CHAN<type> ?
        - readable
      - Then CHAN<type> !
        - Writeable
    - Synchronous communication
      - Send waits for recv and reverse
- Very similar to google's 'go' language
Example: occam

-- foo produces an int on channel 'out'
PROC foo (CHAN INT out !)
  SEQ
    out ! 42  -- write an integer to the channel

-- bar produces bytes on channel screen and reads
-- one int from channel 'in'
PROC bar (CHAN BYTE screen !, CHAN INT in ?)
  INT v:
  SEQ
    screen ! 'a'  -- produce sth on stdout
    screen ! 10   -- newline
    in ? V  -- read the integer that foo wrote

-- execution starts here, keyboard=stdin, screen=stdout, error=stderr
PROC main (CHAN BYTE keyboard?, screen !, error !)
  CHAN INT ch: -- declare a channel that holds at most one integer

PAR
  foo(ch !)     -- this runs
  bar(screen, ch ?)  -- in parallel to this
Example: occam: semantics

```
PROC foo (CHAN BYTE screen!, CHAN INT out !)
  SEQ
    screen ! 'b'
    screen ! 10
    out ! 42
:

PROC bar (CHAN BYTE screen!, CHAN INT in ?)
  INT v:
  SEQ
    screen ! 'a'  -- Error: both foo and bar write to 'screen' at the same time!
    screen ! 10
    in ? v
:

PROC echoing (CHAN BYTE keyboard?, screen!, error!)
  CHAN INT ch:

  PAR
    foo(screen, ch !)
    bar(screen, ch ?)
:
```
Example: occam

```occam

PROTOCOL MSGFORMATS
CASE
  msg0
  msg1; BYTE; BYTE; INT
:

PROC writer(CHAN MSGFORMATS out!)
SEQ
  out ! msg0
  SEQ i = 0 FOR 10
      out ! msg1; 1; (BYTE i) + 1; i
:

PROC reader(CHAN BYTE screen!, CHAN MSGFORMATS in?)
WHILE TRUE
  in ? CASE
    msg0
      SEQ
      screen ! 'X'
      screen ! 10
    
    BYTE x, y:
    INT num:
    msg1; x; y; num
    SEQ
      screen ! 'Z'
      screen ! 10
:

PROC simple.example(CHAN BYTE keyboard?, screen!, error!)
CHAN MSGFORMATS ch:
SEQ
  screen ! 'Y'
  screen ! 10
PAR
  writer(ch !)
  reader(screen, ch ?)

Note: occam allows '.' in names
Note: MSGFORMAT declares two message types that can be sent over a channel
Note: simple.example starts reader and writer running in parallel
Note: writer sends msg0 and 10 x msg1
Note: reader reads msg0 and 10 x msg1
Note: writer exits
Note: readers still wants data
  DEADLOCK for reader process
```

Principles Of Programming Languages
Naming the sender and receiver

- By explicit process name
  - Send message M to process named 'XYZ'
  - As in pure CSP
- By I/O object
  - Send message M over stream S
  - Like occam
    - Allow only one receiver to a channel
    - Concurrent sends not allowed (see example)
  - I/O object that allows multiple senders/receivers
    - “mailbox”
  - More flexible than direct naming
    - Allows modular communication / abstract-data-types
Synchronous vs ASynchronous

- **Sync-send/recv:**
  - wait for message receipt
  - $\text{Wait} = \text{delay} = \text{slow}$
  - $\text{Wait} = \text{simpler/correctness encouraging}$
  - **Sync-recv:** can only wait for fixed set/singleton message
  - Occam, Ada, Banyan (async send, sync recv)

- **Async**
  - **send:** immediately continue with next statement
  - **recv:** tests if message pending, if not continues with error message
  - SR, Hermes: support both sync+async
Explicit or implicit message receipt?

- Most common: explicit receipt
  - Occam, CSP, GO
- Alternative:
  - Create a new named subprocess within receiver for each message arrival
    - Subprocess is usually called a 'thread'
    - Send foo(M) to R
      - Create a new subprocess at R with M as its argument
One-way vs two-way

- Occam: one way as a channel is either input (?) or output (!)
- Remote Procedure Call (RPC), Rendezvous
  - Two-way: Call a method on a remote machine, get a return value back
  - RPC:
    - Java RMI, Corba, etc
    - Orca
  - Rendezvous
    - Ada
Rendezvous

• Ada Tasks
  – Methods of which can be asked to be invoked remotely
    • Called 'entry' methods
      – “entry Get(Line: out Line_Type);”
    – Ada task 'feels' like a remote object
      • Call a method of it: 'entry call'
      • Client calls method on task as normal (and blocks)
        – buffer.Get(L);
      • Receiving process explicitly asks for the next entry call:

-- Accept statement: accept Get(Line: out Line_Type) do
   Line := ..
end Get;
Rendezvous

- Many languages can delay servicing entry calls until wanted
  - arbitrary remote calls are not (immediately) serviced
    - Server can wait until its finished its current job/entry call
    - Server can order the incoming entry calls
  - Enqueued entry calls are "pending"
• Conditional waits for suitable entry call/state

-- Ada:
select
  when <condition1> => accept <entry1(args1)> do ... end <entry1>
  when <condition2> => accept <entry2(args2)> do ... end <entry2>
  when <condition3> => accept <entry3(args3)> do ... end <entry3>
  ...
end select

* When no condition applies, the select statement blocks.
  This is similar to Hoare's CSP's guarded command statement

* the conditions are called 'guards' to the accept statements

* this blocks until one entry call/condition applies
Rendezvous

• Example: (remote) fifo-queue of strings

```
select
  -- when the buffer has something,
  -- we accept remote calls to get a line from the buffer
when Nr_Lines > 0 => accept Get(Line: out Line_Type)
  do
    end Get;
  end
or
  ...
end select
```
task type Buffer_Task is
  entry Put(Line: Line_Type);
  entry Get(Line: out Line_Type);
  entry Stop;
end Buffer_Task;

task body Buffer_Task is
  Max: constant := 10;
  Buffer: array(Integer range 0 .. Max -1) of Line_Type;
  Write_Pos: Integer := 0;
  Read_Pos: Integer := 0;
  Nr_Lines: Boolean := false;
begin
  while not Finished loop
    select
      when Nr_Lines < Max =>
        accept Put(Line: Line_Type) do
          Nr_Lines := Nr_Lines + 1;
          Buffer(Write_Pos) := Line;
          Write_Pos := (Write_Pos + 1) mod Max;
        end Put;
      or when Nr_Lines > 0 =>
        accept Get(out Line: Line_Type) do
          Nr_Lines := Nr_Lines - 1;
          Line := Buffer(Write_Pos);
          Read_Pos := (Read_Pos + 1) mod Max;
        end Get;
      or when Nr_Lines = 0 =>
        accept Stop do
          Finished := True;
        end Stop;
    end select;
  end loop
end Buffer_Task
Rendezvous

• How to use the buffer?
  – Allocate one Buffer, Allocate+start Client process
    • 'Assembler' calls buffer.Get()
  – Allocate+start Server process
    • 'Compiler' calls buffer.Put()

• Entries in the task run mutually exclusive to each other (like a monitor)
  – get/put therefore do not interfere with each other
  – Ada: can't write a guard that depends on the arguments of the entry call
    • "when c > 0 accept foo(c: Integer)" not allowed
Ada: RendezVous extensions

- has extensions to timeout on waiting for server accept

  - Reuses 'select' statement

```ada
select
    when Nr_Lines < Max =>
        accept Put(Line: Line_Type) do
            ....
            end Put;
    or when Nr_Lines > 0 =>
        accept Get(out Line: Line_Type) do
            ...
            end Get;
    or when Nr_Lines = 0 =>
        accept Stop do
            ....
            end Stop;
    after 10  Seconds =>
        ....
    end after;
end select;
```
Ada: protected objects

- Lightweight object
  - Task = protected object + control-flow
  - 'passive' object
- Encapsulate shared data between tasks
  - Ada 9x

```ada
protected type Int_Object is
  function Value return Integer;
  procedure Assign(V:Integer);
  entry Await_Zero;
private
  X: Integer;
end;

protected body Int_Object is
  -- functions may not change the object
  function Value return Integer is
    begin return X; end;
  -- procedures can change state
  procedure Assign(V:Integer);
    begin X:= V; end;
  -- entry call applicable when X = 0
  entry Await_Zero when X = 0 is
    begin null; end;
end;
```
Remote Procedure Calls

- Two way message passing
  - Wait for return value
  - Between different machines
  - RPC looks like a local procedure call
    - Call by value ISO call by reference
    - Can't (in general) use global variables
    - Can't (in general) use pointers
  - Lots of attempts to make them look local
    - Copy-back semantics
      - After RPC copy arguments back
    - Read-only arguments only to RPCs
    - Serialization of arguments to allow pointers
      - Copy arguments (recursively) to a byte-array
      - Recreate data structures at receiver from byte-array
Remote Procedure Calls

- Packing and unpacking of parameters
  - “marshalling” and “unmarshalling”
- The server exports an interface of remotely callable procedures
  - “Service”
- Many RPCs can run at the same time
  - Need concurrency control
    - Monitors
    - Locks
    - etc
Remote Procedure Calls

-- 1) send msg (buffer, “Get()”) to machine holding buffer
-- 2) wait for return value
-- 2a) at remote side create a new thread to invoke Get() in
-- NOTE this is different from RendezVous: server
-- does not 'accept' --> message receipt is implicit
-- NOTE Get uses locks/semaphors/monitors to guard against
-- multiple clients
-- 3) receive a Line_Type back
Line := buffer.Get();

-- 1) pack Line to a byte-array
-- 2) send byte-array to machine holding buffer
-- 3) wait for return value
-- 4) receive a 'None' back
buffer.Put(Line);
Erlang

• Properties:
  - Distributed: message sends with explicit receive
  - fault-tolerant soft real-time
  - non-stop applications: change/update running code
  - Functional basis with pattern matching and modules
    - Prolog like: clauses (=rules) with terms
    - Pattern-matching: condition -> action

```
-module(tut1).
-export([fac/1, mult/2]).

fac(1) -> 1;
fac(N) -> N * fac(N - 1).
mult(X, Y) -> X * Y.

-module(math).
-export([fac/1]).

fac(N) when N > 0 -> N * fac(N-1);
fac(0) -> 1.
```
Erlang

- Datastructures:
  - Atoms
    - A.K.A. Symbolic constants
  - Tuples
  - List

```erlang
-module(atom).
-export([atak/1, foo/1]).

% note: ~n is newline, ~w is an erlang term
atak(name1) -> io:format("name1 taken~n");
atak(name2) -> io:format("name2 taken~n").

% {a,b,c} marks a tuple
foo({name1,N}) -> io:format("alt1 taken ~w~n", [N]);
foo({name2,N}) -> io:format("alt2 taken ~w~n", [N]).
```

1> c(atom). % compile
{ok,atom}
2> atom:atak(name1).
name1 taken
ok
3> atom:foo({name2,1234}).
alt2 taken 1234
ok
Erlang

- Lists are represented using
  - HD | TL
    - HD (one element)
    - TL (the rest of the list)
  - List patterns can be nested:
    - HD1, HD2, HD3, HD4, HD5
    - HD1, HD2, HD3 | TL
  - Assignment can cause pattern-matching

- module(ass).
- export([assign1/0, assign2/0]).

assign1() ->
  [E1, E2, E3] = [1, 2, 3],
  io:format("E1 = ~w~n", [E1]),
  io:format("E2 = ~w~n", [E2]),
  io:format("E3 = ~w~n", [E3]).

assign2() ->
  [E1, E2 | E3] = [1, 2, 3, 4, 5, 6],
  io:format("E1 = ~w~n", [E1]),
  io:format("E2 = ~w~n", [E2]),
  io:format("E3 = ~w~n", [E3]).

9> ass:assign1().
E1 = 1
E2 = 2
E3 = 3
ok

10> ass:assign2().
E1 = 1
E2 = 2
E3 = [3, 4, 5, 6]
ok
Erlang is functional

- Create functions and pass them to other functions / use them in clauses

```erlang
-module(map).
-export([double_mapper/0]).

% 1st argument is a function, 2nd the list
mapper(FunctionArg, [First|Rest]) ->
    [FunctionArg(First) | mapper(FunctionArg,Rest)];

% nothing to do if list is empty
mapper(FunctionArg, []) -> [].

double_mapper() ->
    Doubler = fun(X) -> X * 2 end, % declare a function
    L = [1,2,3,4,5,6,7], % declare a list
    mapper(Doubler,L). % apply the function to the list
```
Erlang

- Distributed execution via processes
  - Processes share no data
  - Created via 'spawn'

```
-module(spawner).
-export([start/0, say_something/2]).

say_something(_, 0) -> done;
say_something(What, Times) ->
    io:format("~w~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(spawner, say_something, [hello, 3]),
    spawn(spawner, say_something, [goodbye, 3]).
```

1> c(spawner).
{ok,spawner}
2> spawner:start().
hello
goodbye
<0.38.0>
hello
goodbye
goodbye
Erlang

- Message sends to a process-id
  - PID ! Term
    - Pid is the return value of a spawn
      - Self() delivers your own PID
    - Term can be an atom, tuple or list

- Message receipt via an blocking receive:
  - receive
    - pattern1 -> actions1;
    - pattern2 -> actions2;
    - ....
  - end.
Erlang

• Example: ping-pong a message a few times

```erlang
-module(ponger).
-export([start/0, pong/0, ping/2]).
% when counter is '0' we send the 'finished' atom
ping(0, Pong_PID) -> Pong_PID ! finished,
    io:format("ping finished\n", []);
ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()}, % send ping
    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} ->
            Pong_PID ! pong, % send pong atom back
            pong() % lets do this again
                % (we have no 'while loop to do this)
    end.
start() ->
    Pong_PID = spawn(ponger, pong, []), % spawn the ping process passing the pong-PID
    spawn(ponger, ping, [3, Pong_PID]).
```

Send !

3> c(ponger).
    {ok,ponger}
4> ponger:start().

Pong received pong
Pong received pong
Ping received pong
Ping received pong
Pong received pong
Pong received pong
Pong finished
Pong finished
Erlang

- Enhanced robustness
  - Receive normally waits forever: Use a timeout clause
  - process-links
    - A separate communication channel for meta-messages
      - Process exit / abnormal termination / exceptions

```erlang
pong() ->
  receive
    {ping, Ping_PID} ->
      io:format("Pong received ping~n", []),
      Ping_PID ! pong,
      pong()
  after 5000 ->
    io:format("Pong timed out~n", [])
  end.
```

```erlang
ping(N, Pong_Pid) ->
  link(Pong_Pid),
  ping1(N, Pong_Pid).
```
Next week: more OO+parallel