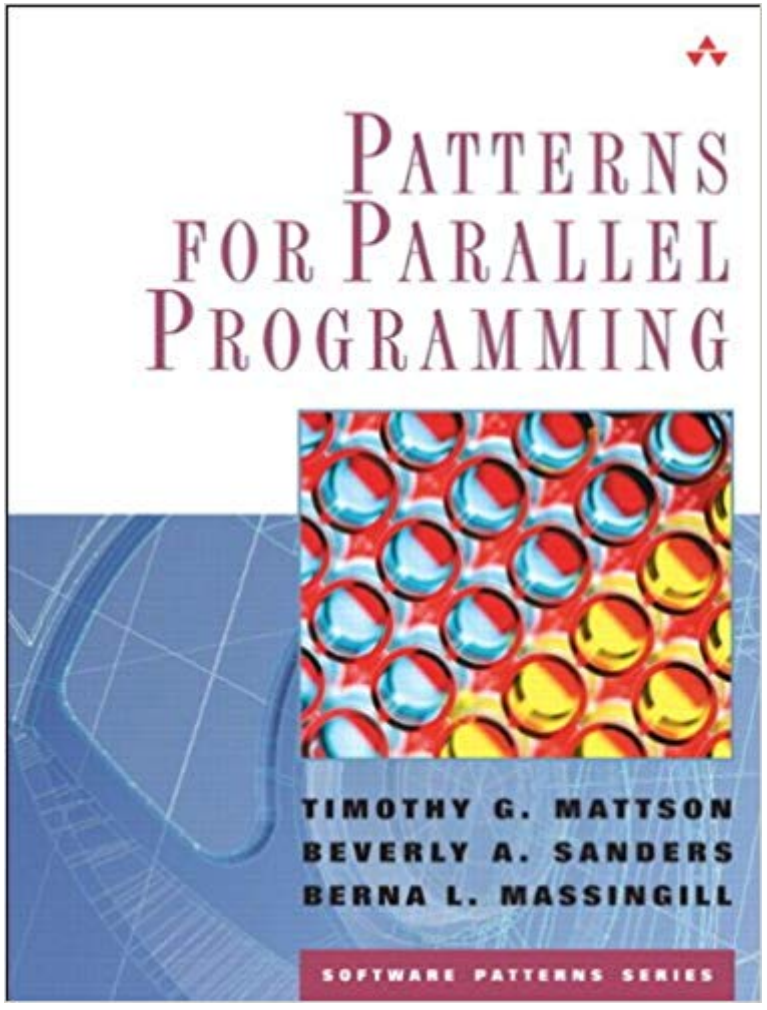
Comparing Common Concurrency Patterns

in Elixir and Erlang

A Pattern Language for Parallel Application Programming

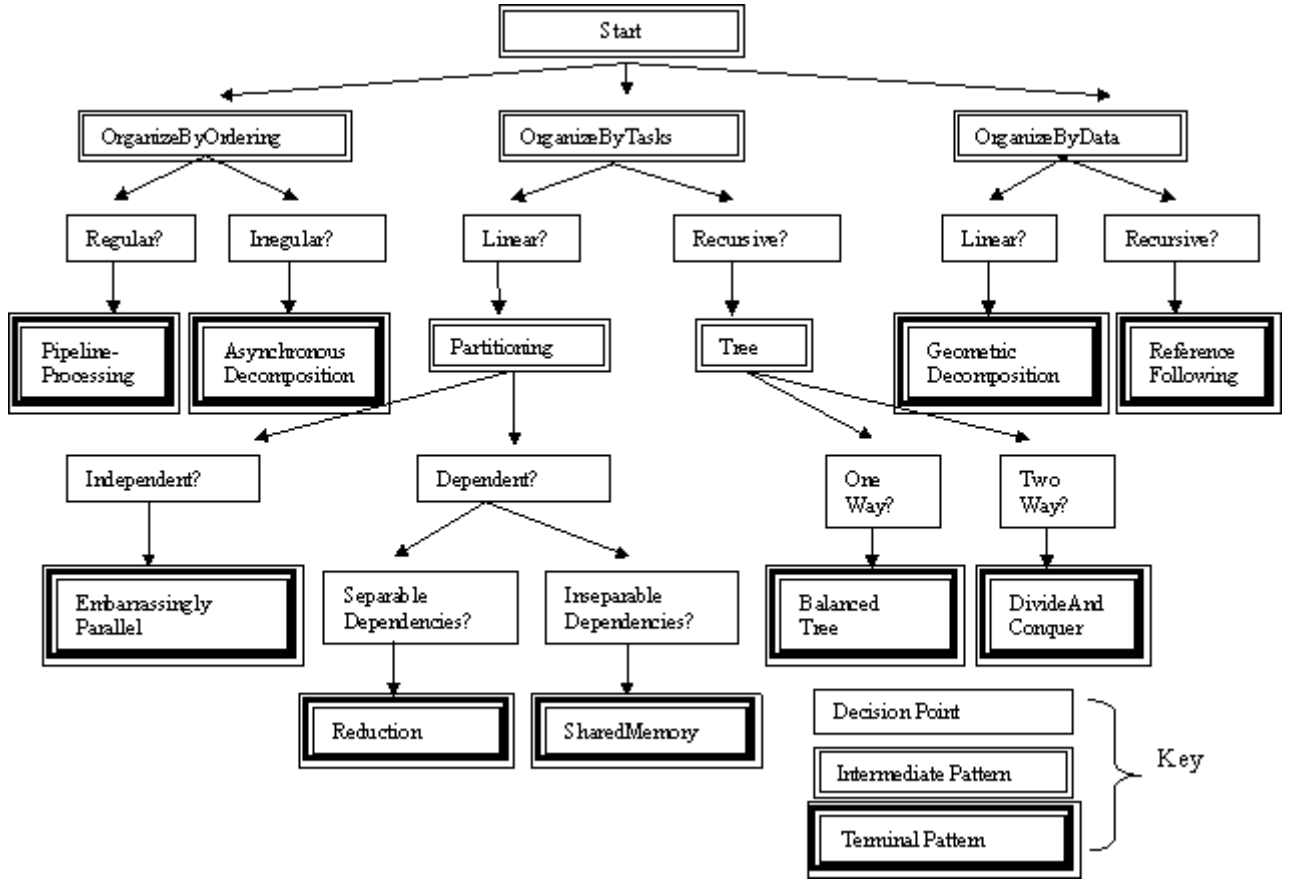
Berna L. Massingill, Timothy G. Mattson, Beverly A. Sanders

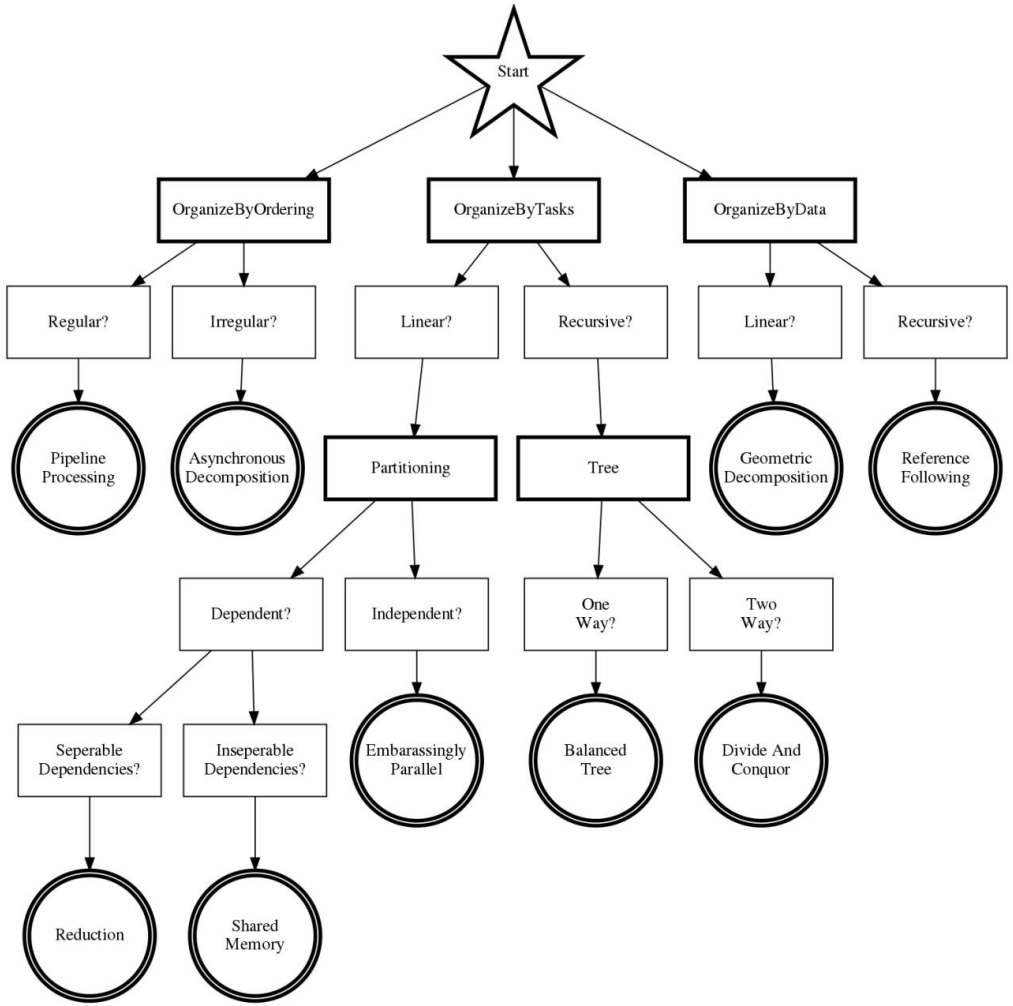

PATTERNS
FOR PARALLEL
PROGRAMMING
TIMOTHY G. MATTSON
BEVERLY A. SANDERS
BERNA L. MASSINGILL
SOFTWARE PATTERNS SERIES

Comparing Common Concurrency Patterns

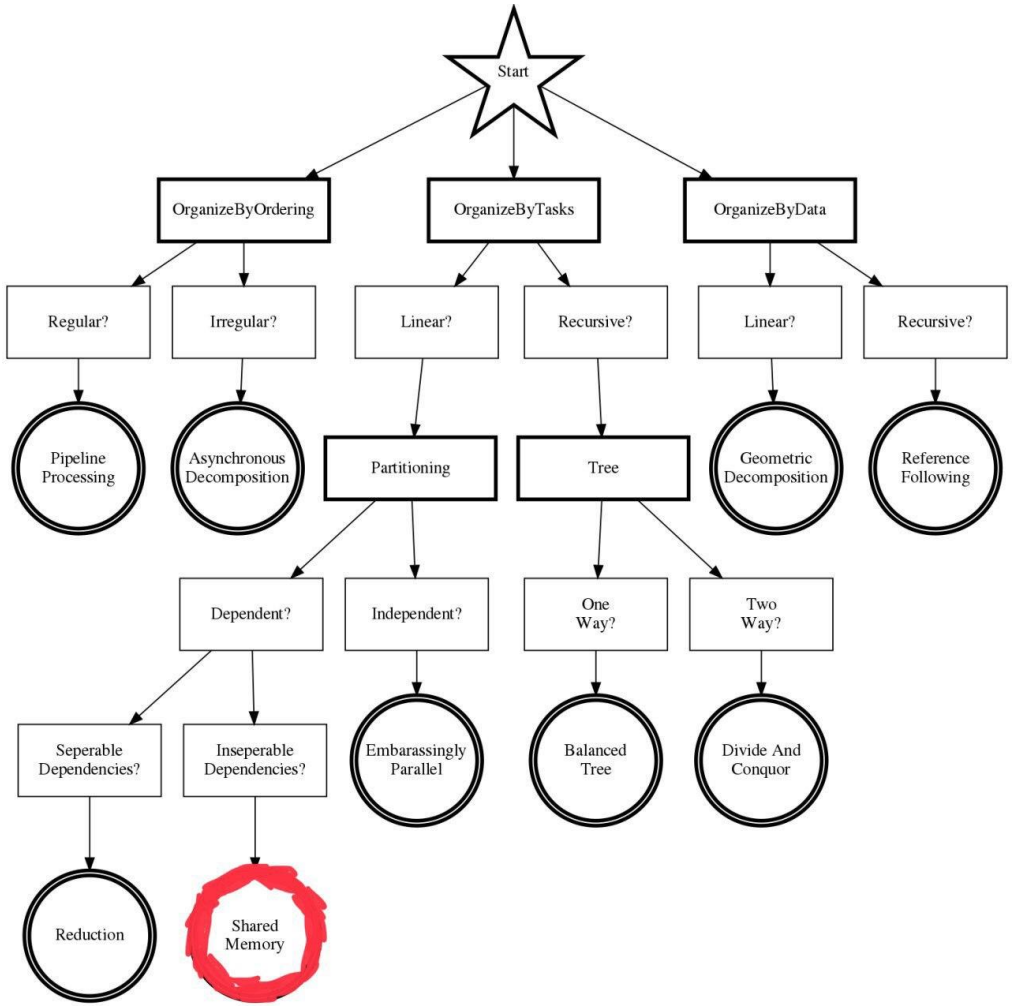
in Elixir and Erlang

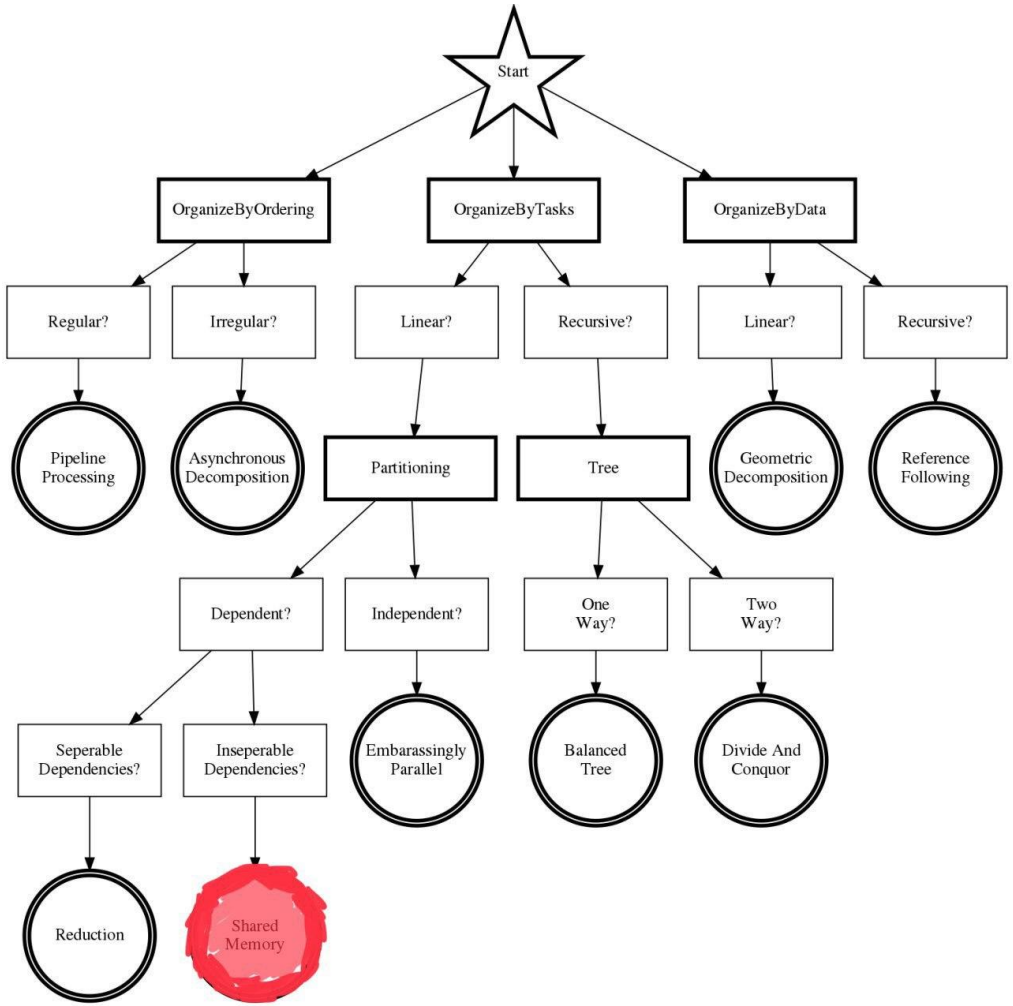

CCCP
FETISOV

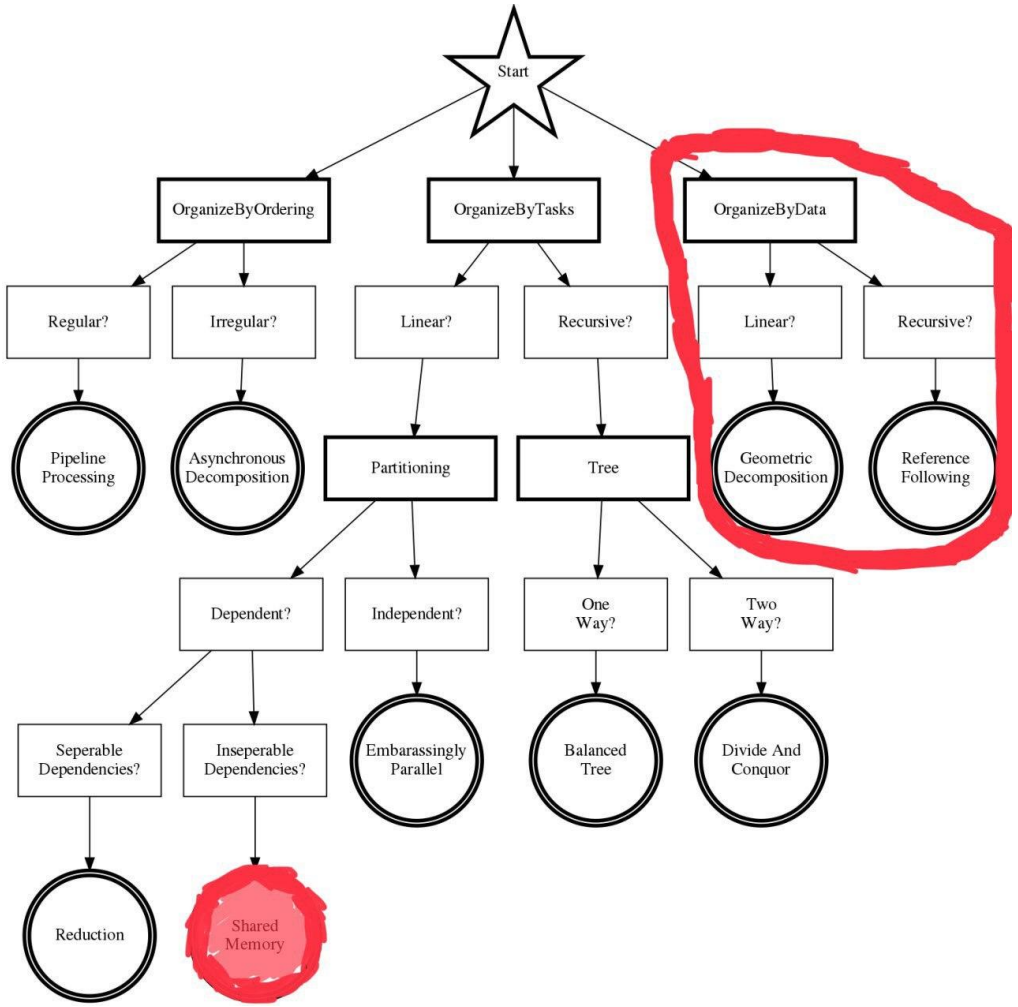

Start
OrganizeByOrdering
OrganizeByTasks
OrganizeByData
Regular?
Irregular?
Linear?
Recursive?
Linear?
Recursive?
Pipeline-Processing
Asynchronous Decomposition
Partitioning
Tree
Geometric Decomposition
Reference Following
Independent?
Dependent?
One Way?
Two Way?
Embarrassingly Parallel
Separable Dependencies?
Inseparable Dependencies?
Balanced Tree
DivideAnd Conquer
Reduction
SharedMemory
Decision Point
Intermediate Pattern
Terminal Pattern
Key

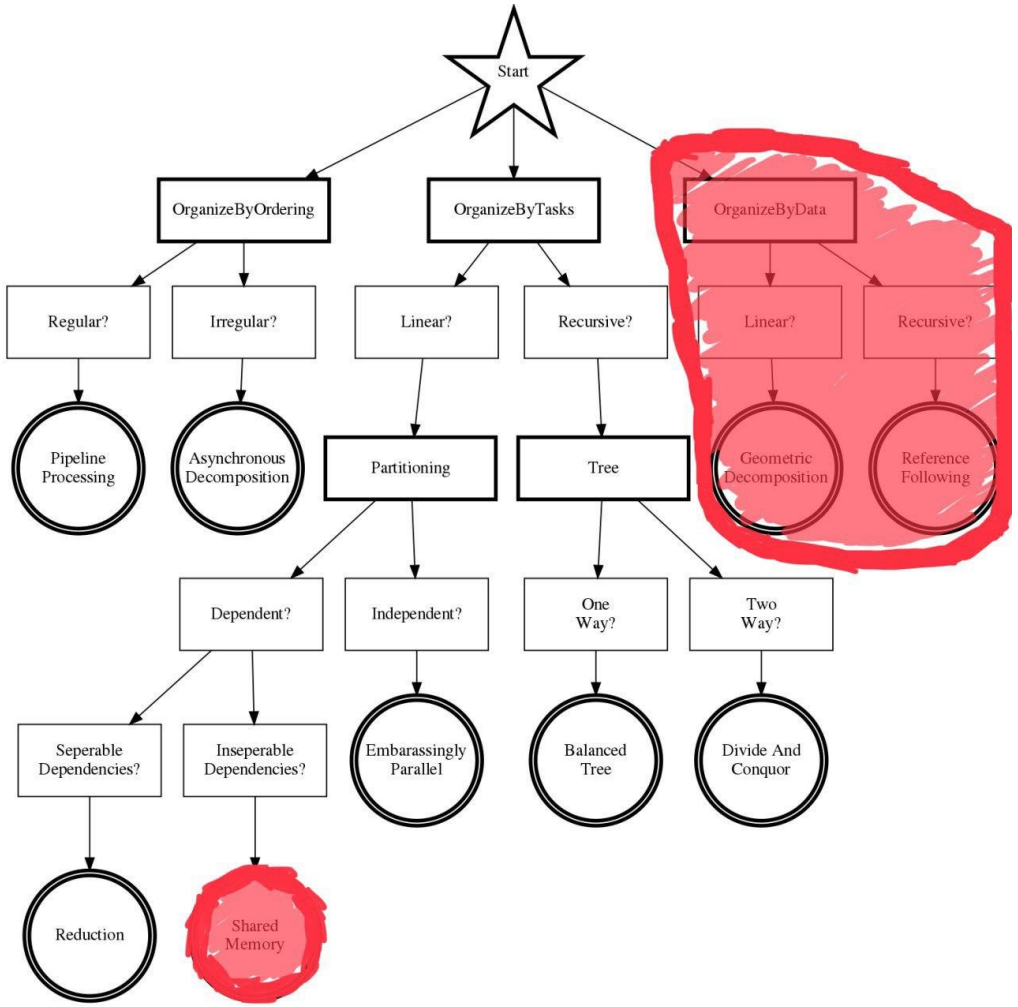

Start
OrganizeByOrdering
OrganizeByTasks
OrganizeByData
Regular?
Irregular?
Linear?
Recursive?
Linear?
Recursive?
Pipeline Processing
Asynchronous Decomposition
Partitioning
Tree
Geometric Decomposition
Reference Following
Dependent?
Independent?
One Way?
Two Way?
Seperable Dependencies?
Inseperable Dependencies?
Embarassingly Parallel
Balanced Tree
Divide And Conquor
Reduction
Shared Memory

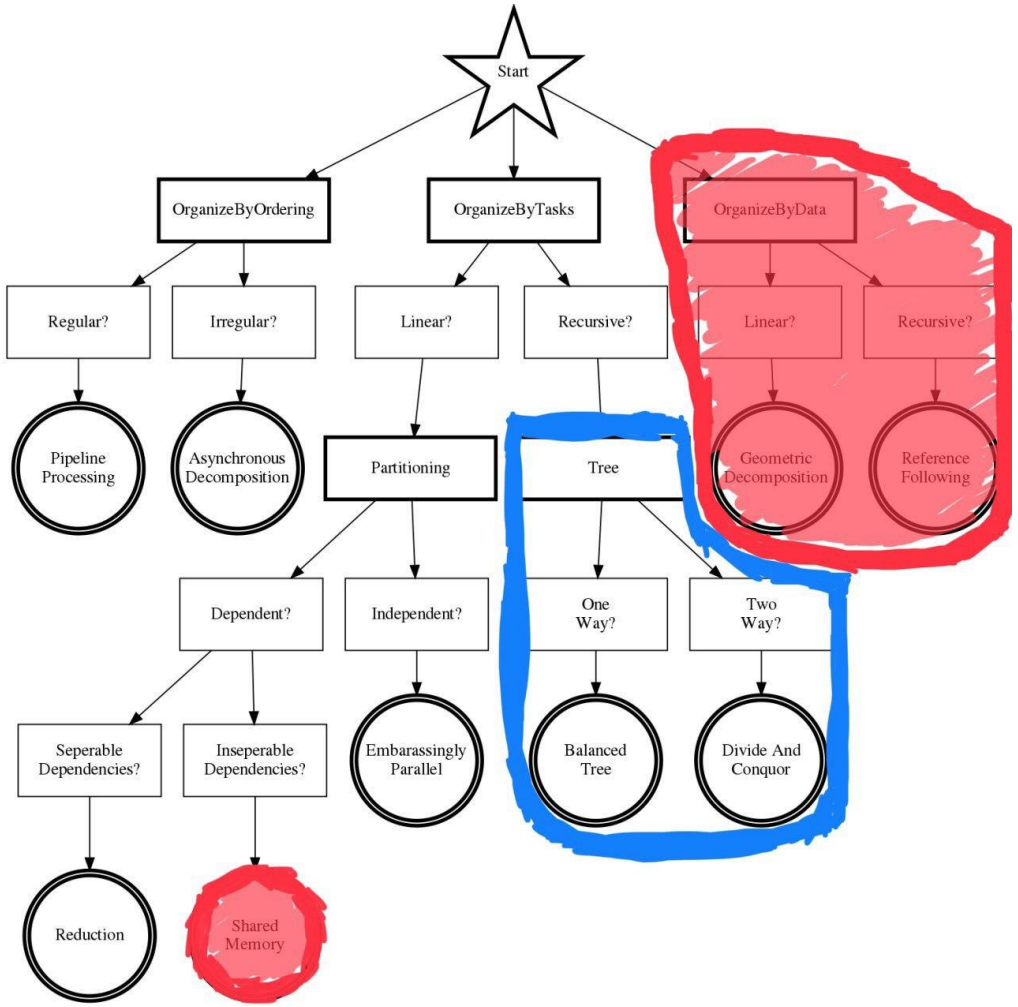

IBM
IBM

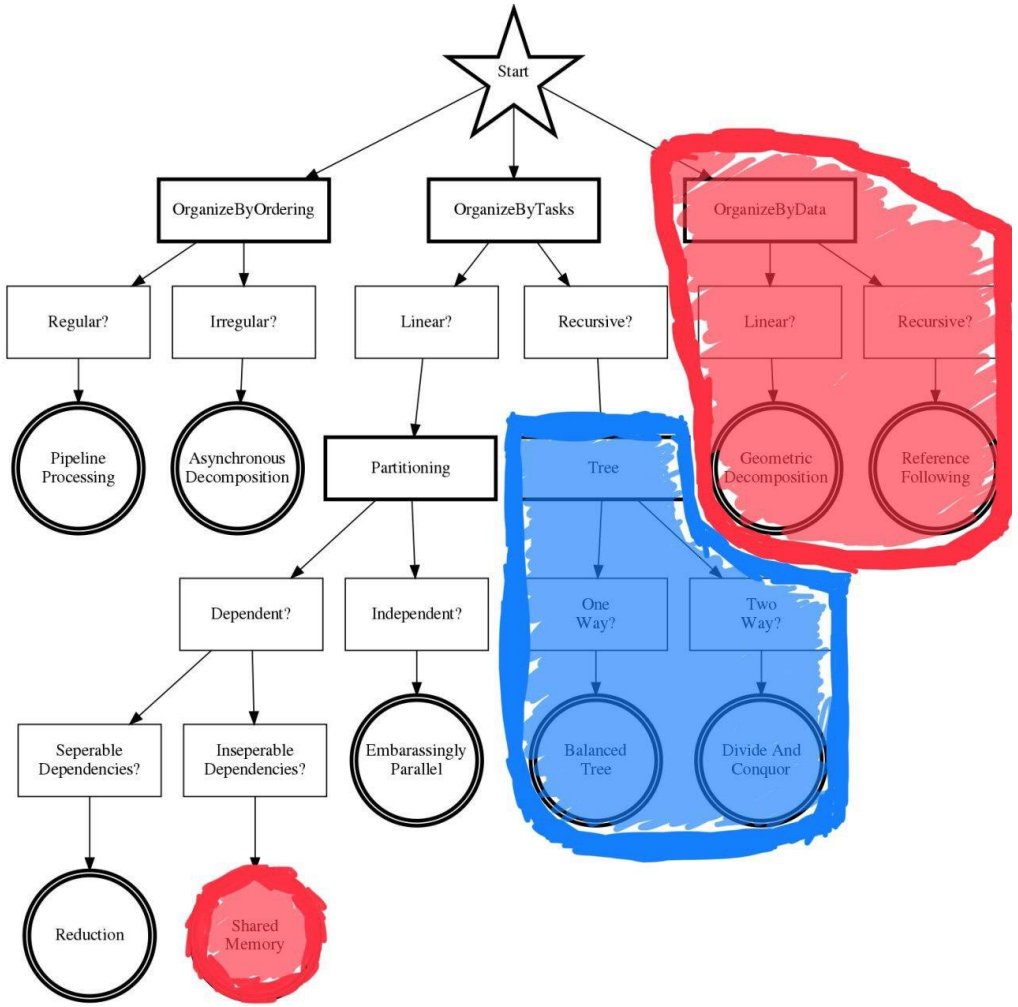

Start
OrganizeByOrdering
OrganizeByTasks
OrganizeByData
Regular?
Irregular?
Linear?
Recursive?
Linear?
Recursive?
Pipeline Processing
Asynchronous Decomposition
Partitioning
Tree
Geometric Decomposition
Reference Following
Dependent?
Independent?
One Way?
Two Way?
Seperable Dependencies?
Inseperable Dependencies?
Embarassingly Parallel
Balanced Tree
Divide And Conquor
Reduction
Shared Memory

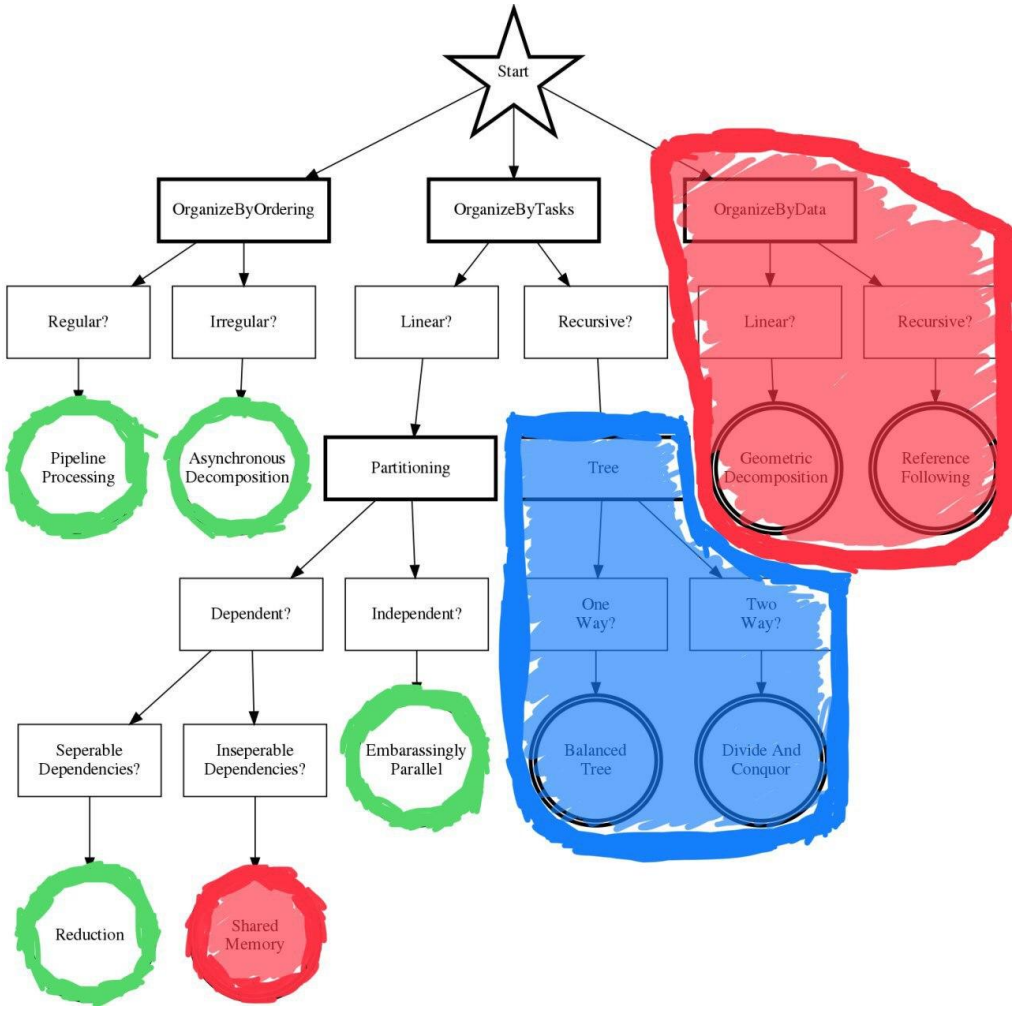

Start
OrganizeByOrdering
OrganizeByTasks
OrganizeByData
Regular?
Irregular?
Linear?
Recursive?
Linear?
Recursive?
Pipeline Processing
Asynchronous Decomposition
Partitioning
Tree
Geometric Decomposition
Reference Following
Dependent?
Independent?
One Way?
Two Way?
Seperable Dependencies?
Inseperable Dependencies?
Embarassingly Parallel
Balanced Tree
Divide And Conquor
Reduction
Shared Memory

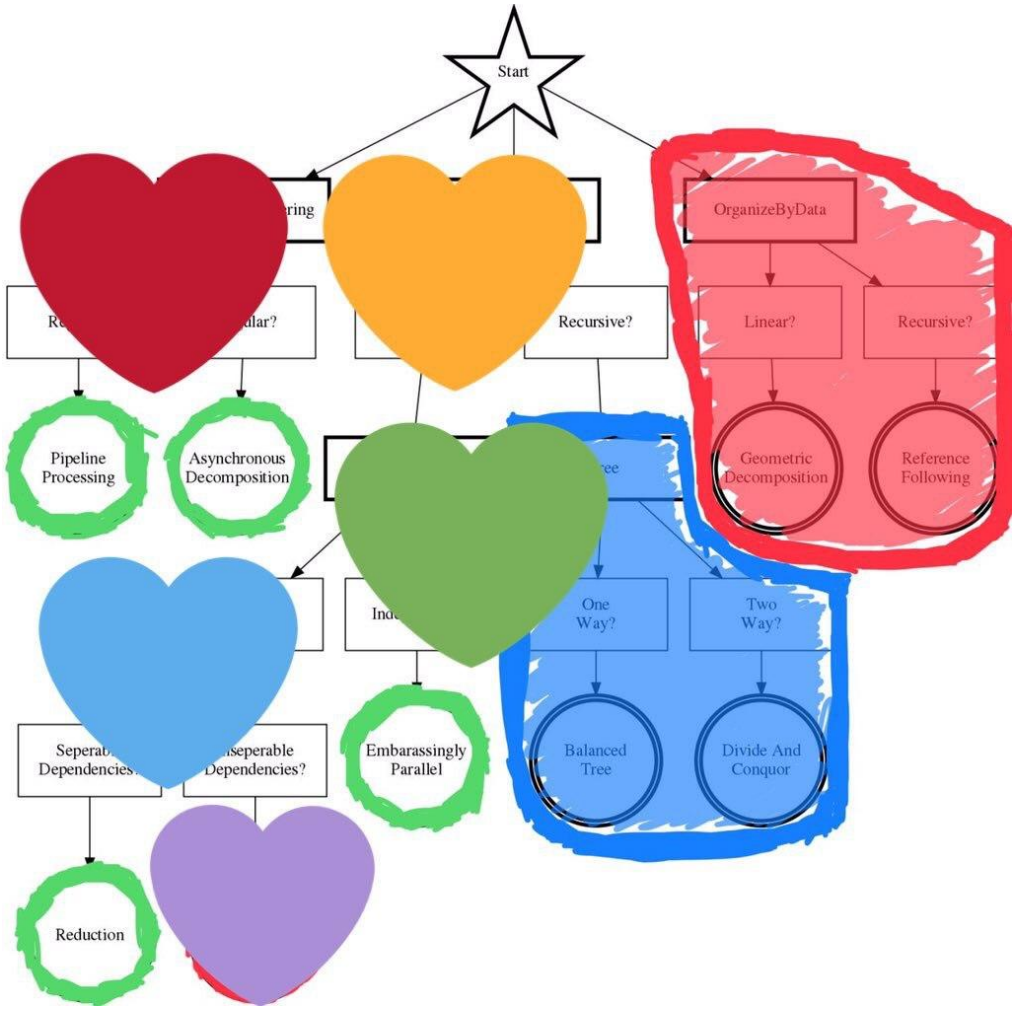

Start
OrganizeByOrdering
OrganizeByTasks
OrganizeByData
Regular?
Irregular?
Linear?
Recursive?
Linear?
Recursive?
Pipeline Processing
Asynchronous Decomposition
Partitioning
Tree
Geometric Decomposition
Reference Following
Dependent?
Independent?
One Way?
Two Way?
Seperable Dependencies?
Inseperable Dependencies?
Embarassingly Parallel
Balanced Tree
Divide And Conquor
Reduction
Shared Memory


Start
OrganizeByOrdering
OrganizeByTasks
OrganizeByData
Regular?
Irregular?
Linear?
Recursive?
Linear?
Recursive?
Pipeline Processing
Asynchronous Decomposition
Partitioning
Tree
Geometric Decomposition
Reference Following
Dependent?
Independent?
One Way?
Two Way?
Seperable Dependencies?
Inseperable Dependencies?
Embarassingly Parallel
Balanced Tree
Divide And Conquor
Reduction
Shared Memory


Start
OrganizeByOrdering
OrganizeByTasks
OrganizeByData
Regular?
Irregular?
Linear?
Recursive?
Linear?
Recursive?
Pipeline Processing
Asynchronous Decomposition
Partitioning
Tree
Geometric Decomposition
Reference Following
Dependent?
Independent?
One Way?
Two Way?
Seperable Dependencies?
Inseperable Dependencies?
Embarassingly Parallel
Balanced Tree
Divide And Conquor
Reduction
Shared Memory


Start
OrganizeByOrdering
OrganizeByTasks
OrganizeByData
Regular?
Irregular?
Linear?
Recursive?
Linear?
Recursive?
Pipeline Processing
Asynchronous Decomposition
Partitioning
Tree
Geometric Decomposition
Reference Following
Dependent?
Independent?
One Way?
Two Way?
Seperable Dependencies?
Inseperable Dependencies?
Embarassingly Parallel
Balanced Tree
Divide And Conquor
Reduction
Shared Memory


Start
OrganizeByOrdering
OrganizeByTasks
OrganizeByData
Regular?
Irregular?
Linear?
Recursive?
Linear?
Recursive?
Pipeline Processing
Asynchronous Decomposition
Partitioning
Tree
Geometric Decomposition
Reference Following
Dependent?
Independent?
One Way?
Two Way?
Seperable Dependencies?
Inseperable Dependencies?
Embarassingly Parallel
Balanced Tree
Divide And Conquor
Reduction
Shared Memory


Start
OrganizeByData
Linear?
Recursive?
Recursive?
Pipeline Processing
Asynchronous Decomposition
Geometric Decomposition
Reference Following
One Way?
Two Way?
Balanced Tree
Divide And Conquor
Embarassingly Parallel
Reduction

Benchee Reduction

Benchee

ExUnit

Reduction

Asynchronous

Decomposition

Benchee	Reduction
ExUnit	Asynchronous Decomposition
GenStage	Pipeline Processing

Embarrassingly Parallel

Embarrassingly Parallel

```
def send_emails(emails) do # serial
  Enum.map(emails, fn email ->
    send_email(email)
  end)
end
```

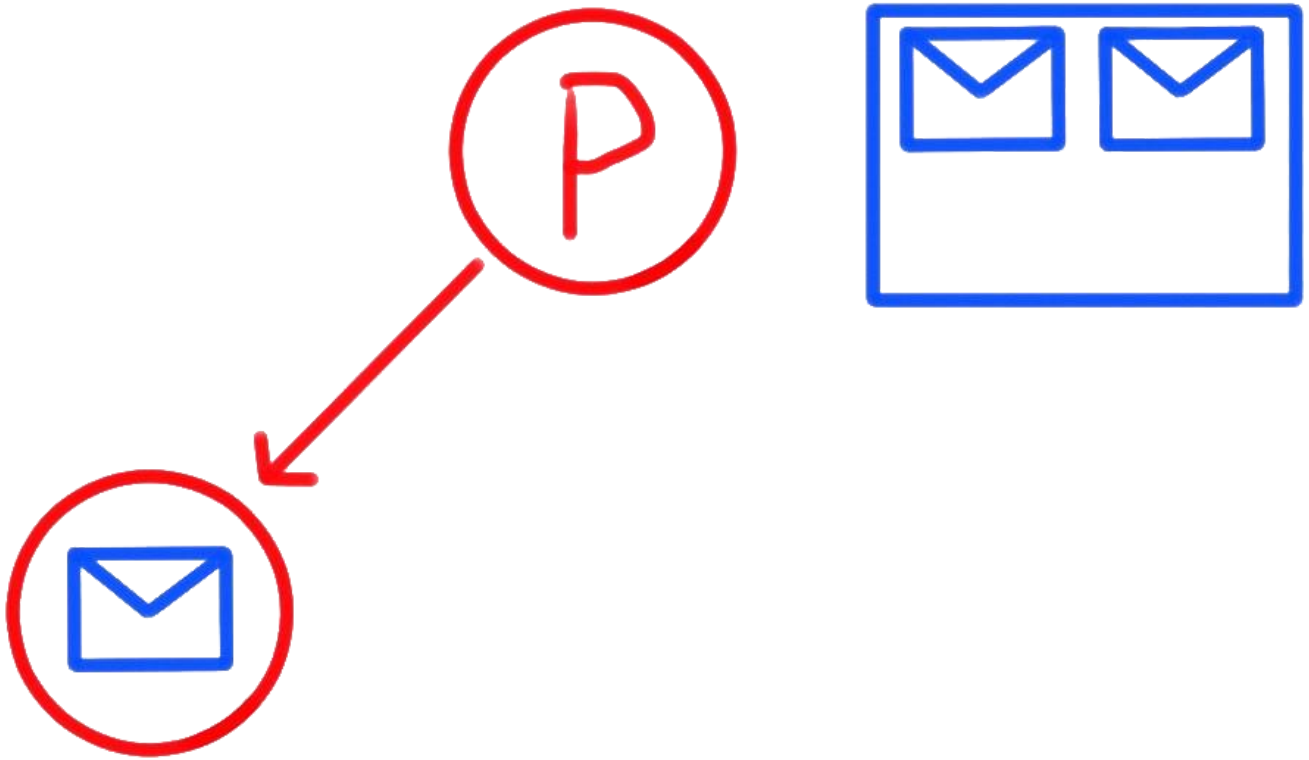
Embarrassingly Parallel

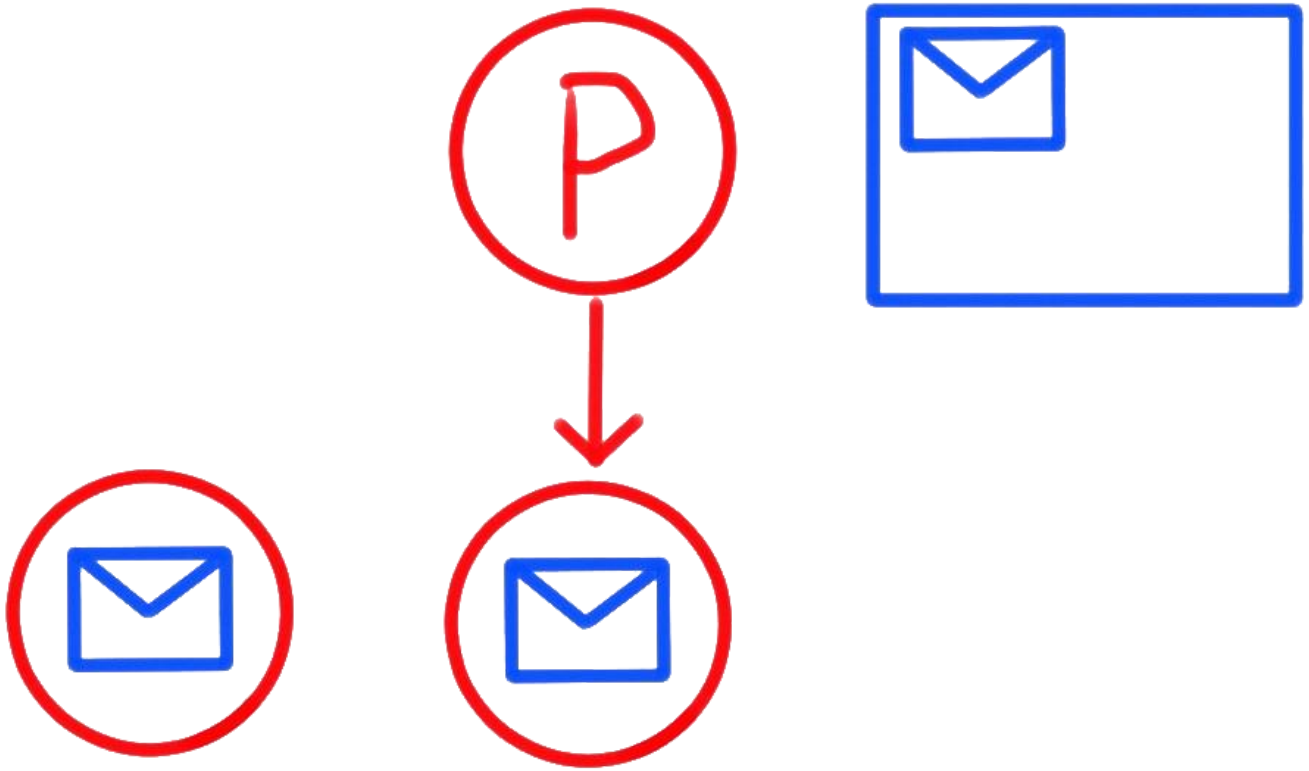
```
def send_emails(emails) do # serial
  Enum.map(emails, fn email ->
    send_email(email)
  end)
end

def send_emails(emails) do # parallel
  Enum.map(emails, fn email ->
    spawn(fn -> send_email(email) end)
  end)
end
```

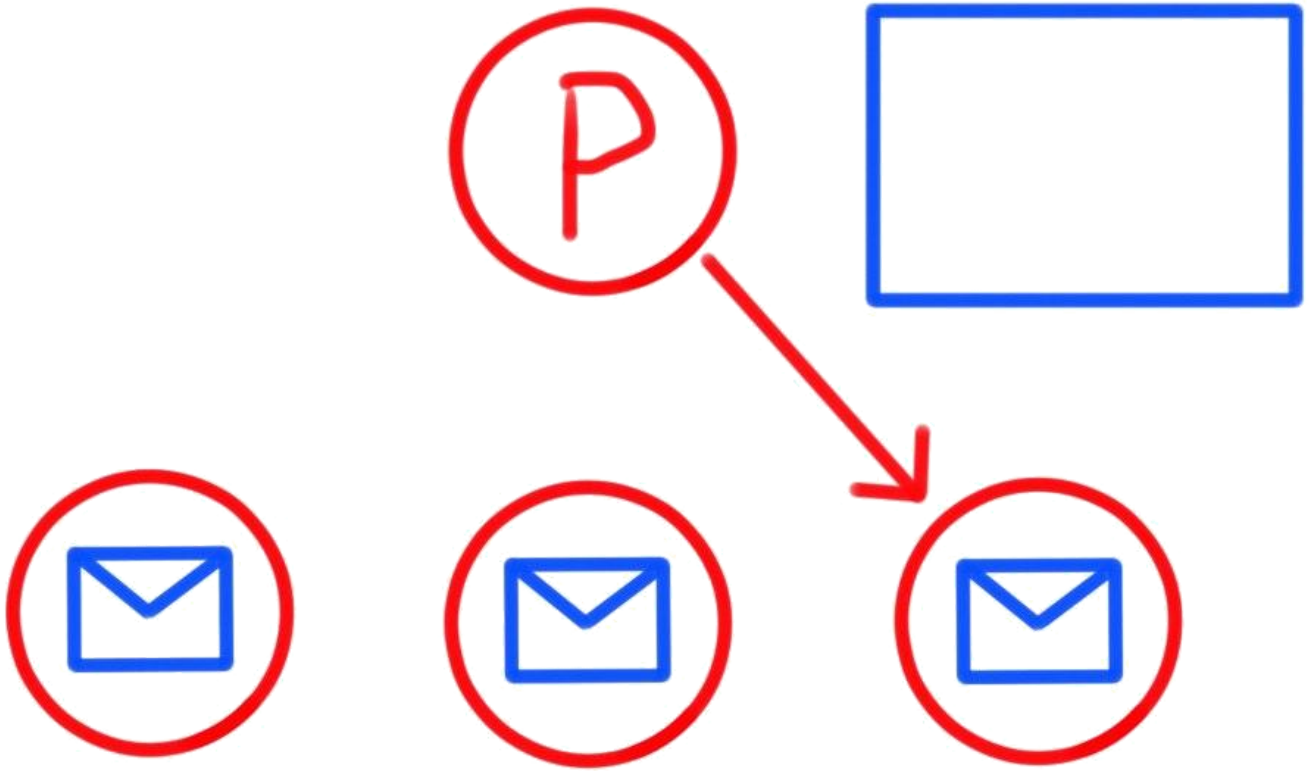
Embarrassingly Parallel

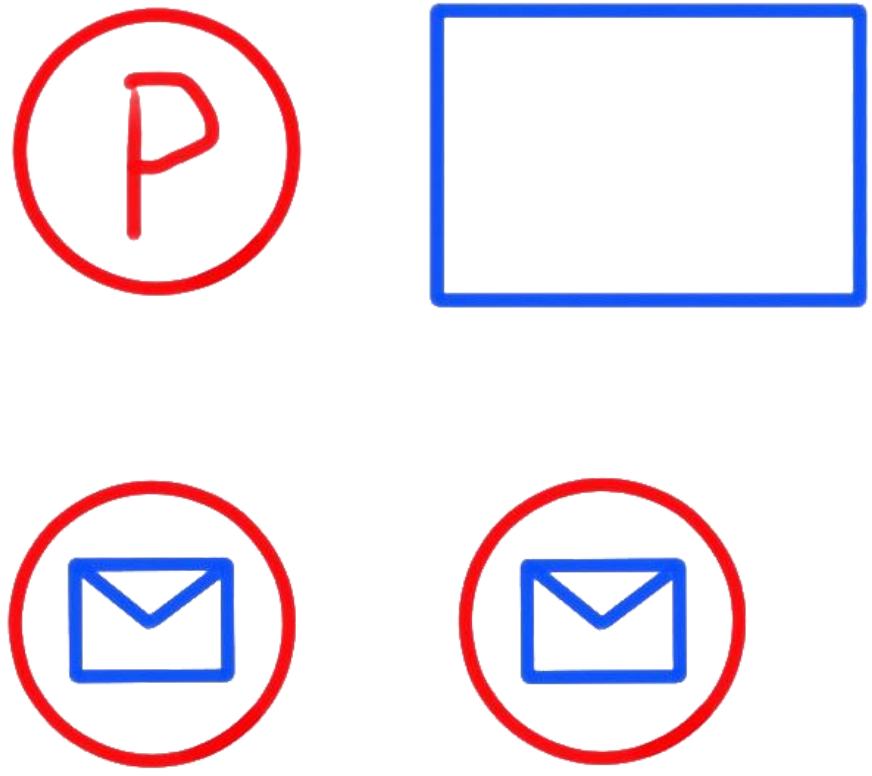
```elixir
def send_emails(emails) do # serial
  Enum.map(emails, fn email ->
    send_email(email)
  end)
end

def send_emails(emails) do # parallel
  Enum.map(emails, fn email ->
    spawn(fn -> send_email(email) end)
  end)
end
```

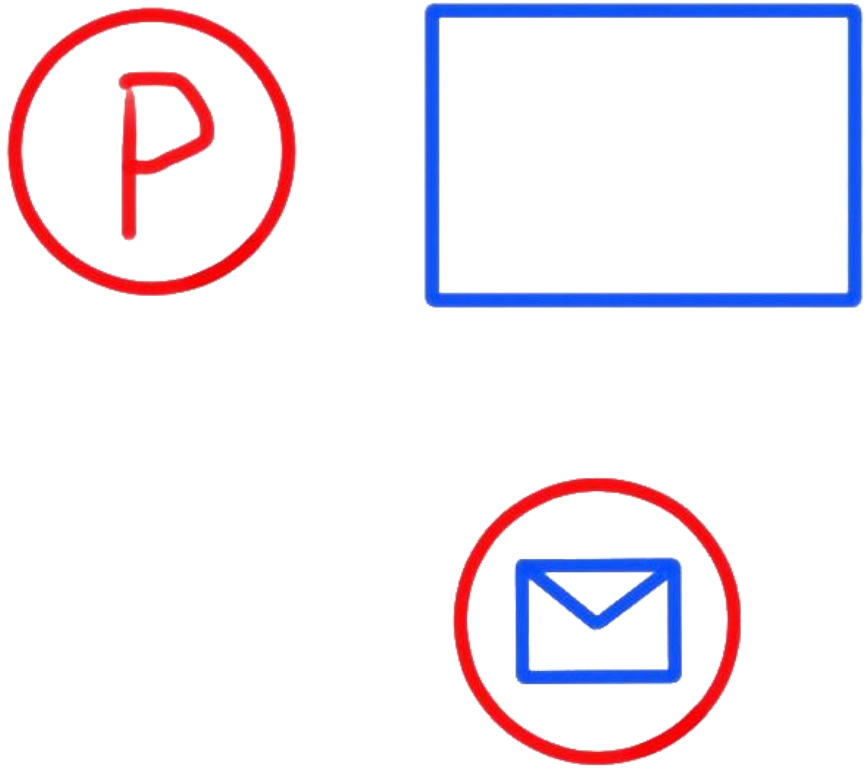

P


P


P


P


P



Done!

Embarrassingly Parallel

- Eagerly evaluated, linear input

Embarrassingly Parallel

- Eagerly evaluated, linear input
- No communication with parent

Embarrassingly Parallel

- Eagerly evaluated, linear input
- No communication with parent
- No communication with siblings

Embarrassingly Parallel

- Eagerly evaluated, linear input
- No communication with parent
- No communication with siblings
- No shared dependencies between siblings

Reduction

Reduction

```elixir
defmodule Parallel do
  def map(collection, func) do
    refs =
      Enum.map(collection, fn element ->
        Task.async(fn -> func.(element) end)
      end)

    Enum.map(refs, fn ref ->
      Task.await(ref, :infinity))
    end)
  end
end
```

Reduction

```elixir
defmodule Parallel do
  def map(collection, func) do
    refs =
      Enum.map(collection, fn element ->
        Task.async(fn -> func.(element) end)
      end)

    Enum.map(refs, fn ref ->
      Task.await(ref, :infinity))
    end)
  end
end
```

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end
```

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end
```

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end
```

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end
```

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end
```

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end


def await(ref, timeout \\ :infinity) do
  receive do
    {^ref, value} -> value
    after timeout -> {:error, :timeout}
  end
end
```

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end


def await(ref, timeout \\ :infinity) do
  receive do
    {^ref, value} -> value
    after timeout -> {:error, :timeout}
  end
end
```

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end


def await(ref, timeout \\ :infinity) do
  receive do
    {^ref, value} -> value
    after timeout -> {:error, :timeout}
  end
end
```

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end


def await(ref, timeout \\ :infinity) do
  receive do
    {^ref, value} -> value
    after timeout -> {:error, :timeout}
  end
end
```

Basic async & await

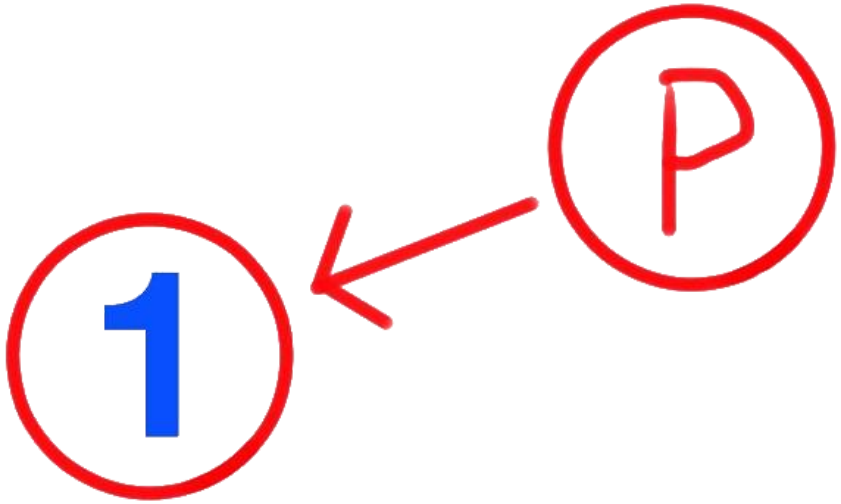
```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end


def await(ref, timeout \\ :infinity) do
  receive do
    {^ref, value} -> value
    after timeout -> {:error, :timeout}
  end
end
```

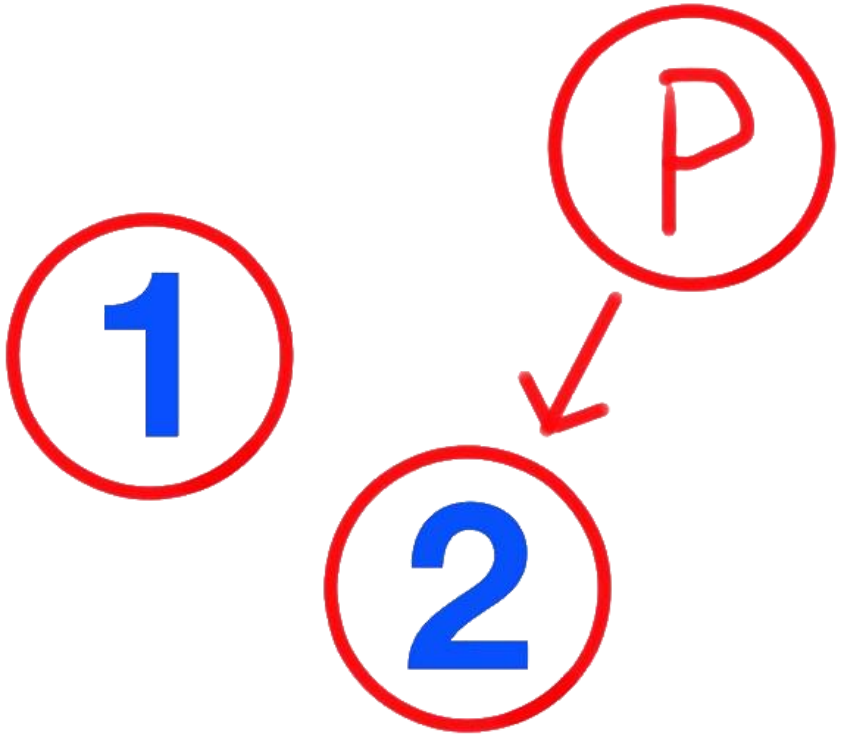
```
me = self()

fun = fn seconds ->
  Process.sleep(seconds * 1000)    # Simulate some expensive work
  send(me, Time.utc_now().second)
end
```

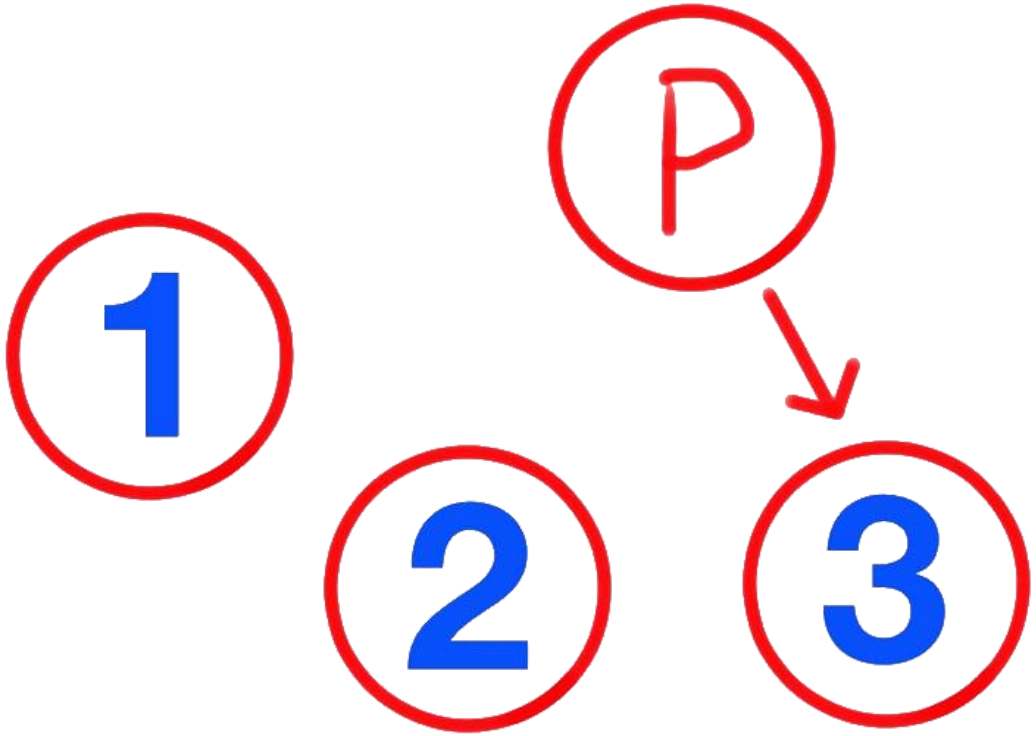
```elixir
me = self()

fun = fn seconds ->
  Process.sleep(seconds * 1000)    # Simulate some expensive work
  send(me, Time.utc_now().second)
end


Time.utc_now().second              #=> 48
```

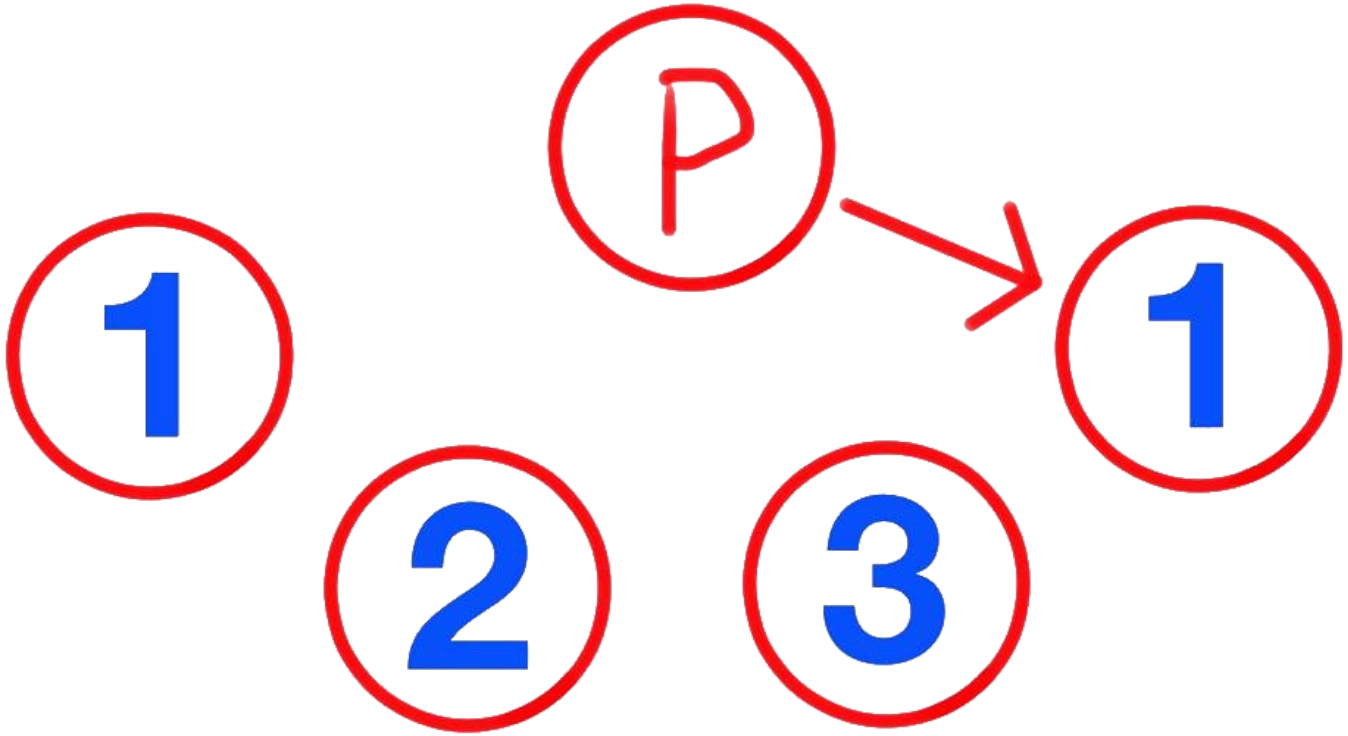
```elixir
me = self()

fun = fn seconds ->
  Process.sleep(seconds * 1000)    # Simulate some expensive work
  send(me, Time.utc_now().second)
end


Time.utc_now().second              #=> 48


Parallel.map([1, 2, 3, 1], fun)    #=> [49, 50, 51, 49]
```

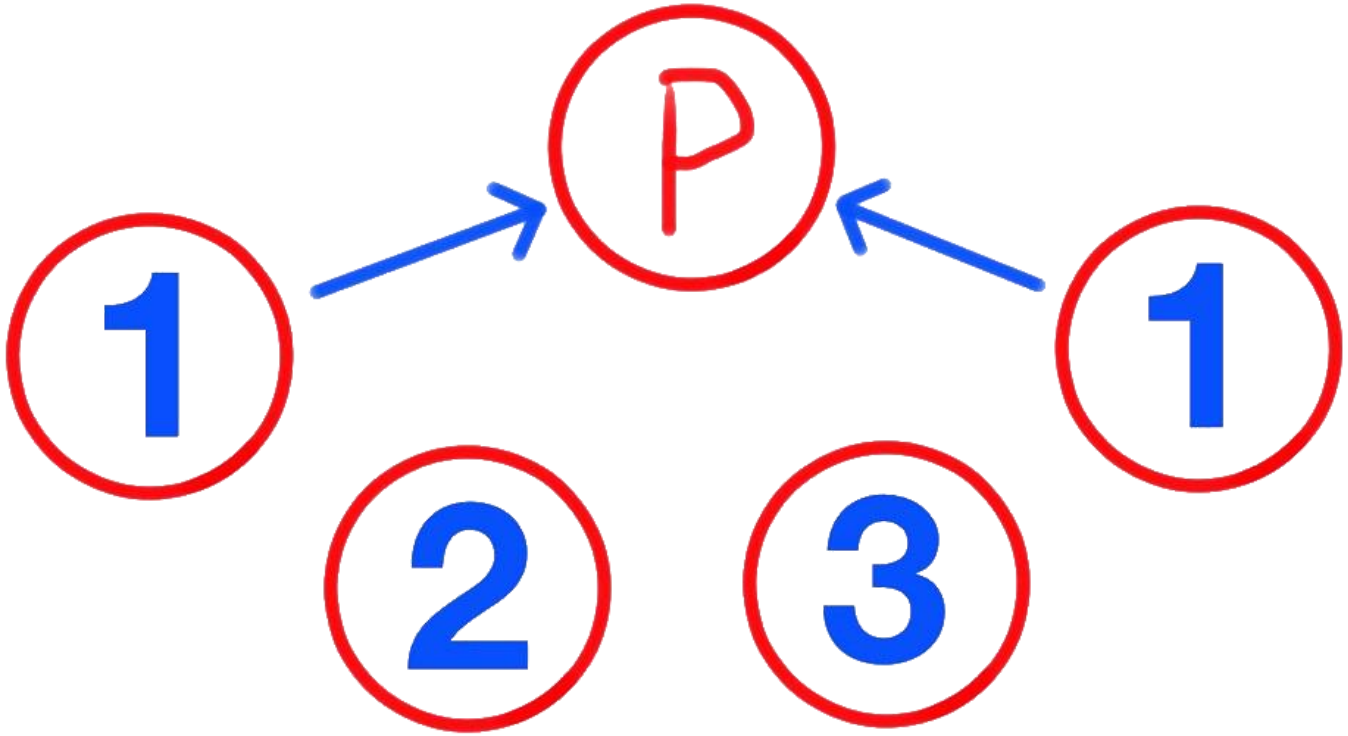
```elixir
me = self()

fun = fn seconds ->
  Process.sleep(seconds * 1000)    # Simulate some expensive work
  send(me, Time.utc_now().second)
end


Time.utc_now().second              #=> 48


Parallel.map([1, 2, 3, 1], fun)    #=> [49, 50, 51, 49]


Time.utc_now().second              #=> 51
```

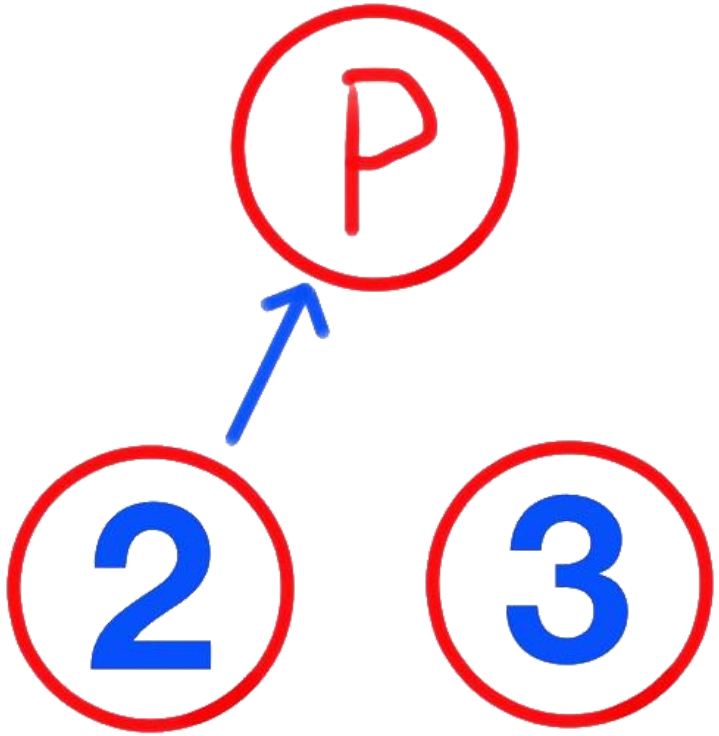

P
1

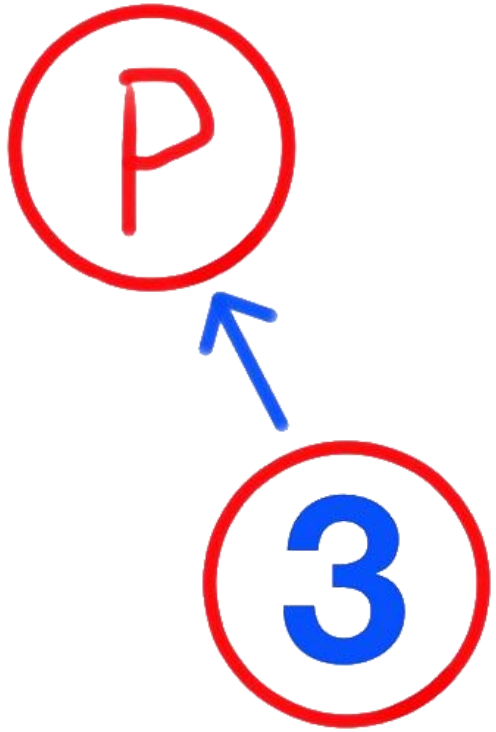

P
1
2


P
1
2
3


P
1
1
2
3


P
1
1
2
3


P
2
3


P
3

Basic async & await

```
def async(func) do
  ref = make_ref()
  me = self()
  spawn(fn -> send(me, {ref, func()}) end)
  ref
end

def await(ref, timeout \\ :infinity) do
  receive do
    {^ref, value} -> value
    after timeout -> {:error, :timeout}
  end
end
```

Distributed Async & Await

```
def async(func, node \\ Node.self()) do
  ref = make_ref()
  me = self()
  Node.spawn(node, fn -> send(me, {ref, func()}) end)
  ref
end

def await(ref, timeout) do
  receive do
    {^ref, value} -> value
    after timeout -> {:error, :timeout}
  end
end
```

Reduction

- Eagerly evaluated, linear input

Reduction

- Eagerly evaluated, linear input
- Communication with parent allowed!

Reduction

- Eagerly evaluated, linear input
- Communication with parent allowed!
- No communication between children

Reduction

- Eagerly evaluated, linear input
- Communication with parent allowed!
- No communication between children
- No shared dependencies between children

Asynchronous Decomposition

Asynchronous Decomposition

```
def run() do
  # ...
end

def loop(:async) do
  # ...
end

def loop(:sync) do
  # ...
end

def run_module(mod) do
  # ...
end

def run_test(test) do
  # ...
end
```

Asynchronous Decomposition

```
def run() do
  EventManager.start()

  EventManager.suite_started()
  loop(:async)
  EventManager.suite_finished()

  results = EventManager.get_results()
  EventManger.stop()
  results
end
```

Asynchronous Decomposition

```
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end
```

Asynchronous Decomposition

```
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end

def loop(:sync) do
  case wait_for_module(:sync) do
    {:ok, mod} ->
      run_module(mod)
      loop(:sync)

    {:error, :no_more_modules} ->
      :ok
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end

def loop(:sync) do
  case wait_for_module(:sync) do
    {:ok, mod} ->
      run_module(mod)
      loop(:sync)

    {:error, :no_more_modules} ->
      :ok
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end

def loop(:sync) do
  case wait_for_module(:sync) do
    {:ok, mod} ->
      run_module(mod)
      loop(:sync)

    {:error, :no_more_modules} ->
      :ok
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end

def loop(:sync) do
  case wait_for_module(:sync) do
    {:ok, mod} ->
      run_module(mod)                    # blocking
      loop(:sync)

    {:error, :no_more_modules} ->
      :ok
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end

def loop(:sync) do
  case wait_for_module(:sync) do
    {:ok, mod} ->
      run_module(mod)                    # blocking
      loop(:sync)

    {:error, :no_more_modules} ->
      :ok
  end
end
```

Asynchronous Decomposition

```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end

def loop(:sync) do
  case wait_for_module(:sync) do
    {:ok, mod} ->
      run_module(mod)                    # blocking
      loop(:sync)

    {:error, :no_more_modules} ->
      :ok
  end
end
```

Asynchronous Decomposition

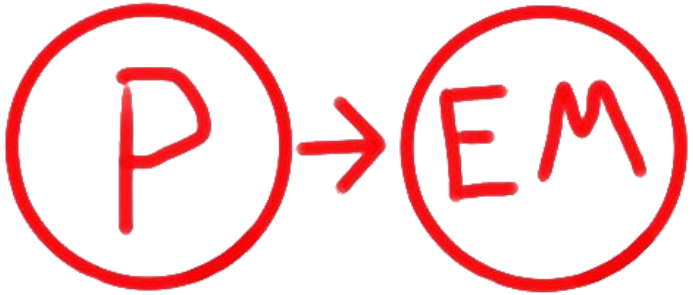
```elixir
def loop(:async) do
  case wait_for_module(:async) do
    {:ok, mod} ->
      spawn(fn -> run_module(mod) end)   # non-blocking
      loop(:async)

    {:error, :no_more_async_modules} ->
      loop(:sync)
  end
end

def loop(:sync) do
  case wait_for_module(:sync) do
    {:ok, mod} ->
      run_module(mod)                    # blocking
      loop(:sync)

    {:error, :no_more_modules} ->
      :ok
  end
end
```

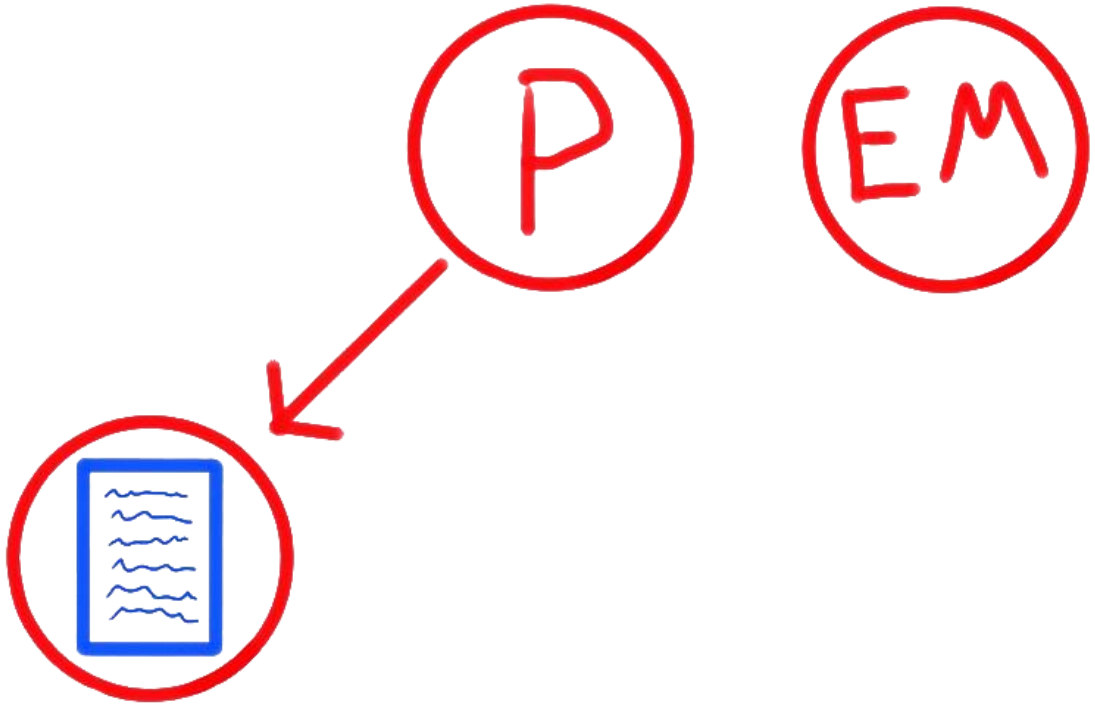
Asynchronous Decomposition

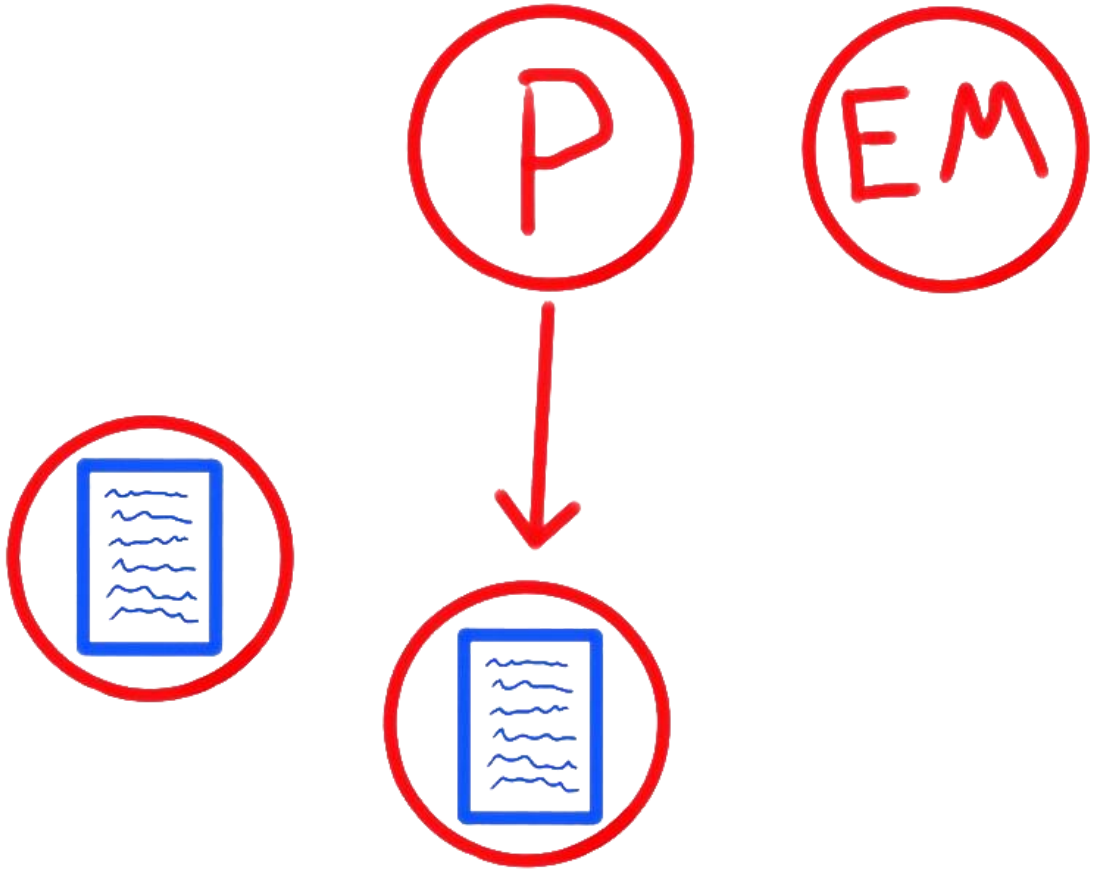
```elixir
def run_module(mod) do
  EventManager.module_started(mod)

  mod
  |> get_tests()
  |> Enum.map(&run_test/1)

  EventManager.module_finished(mod)
end
```

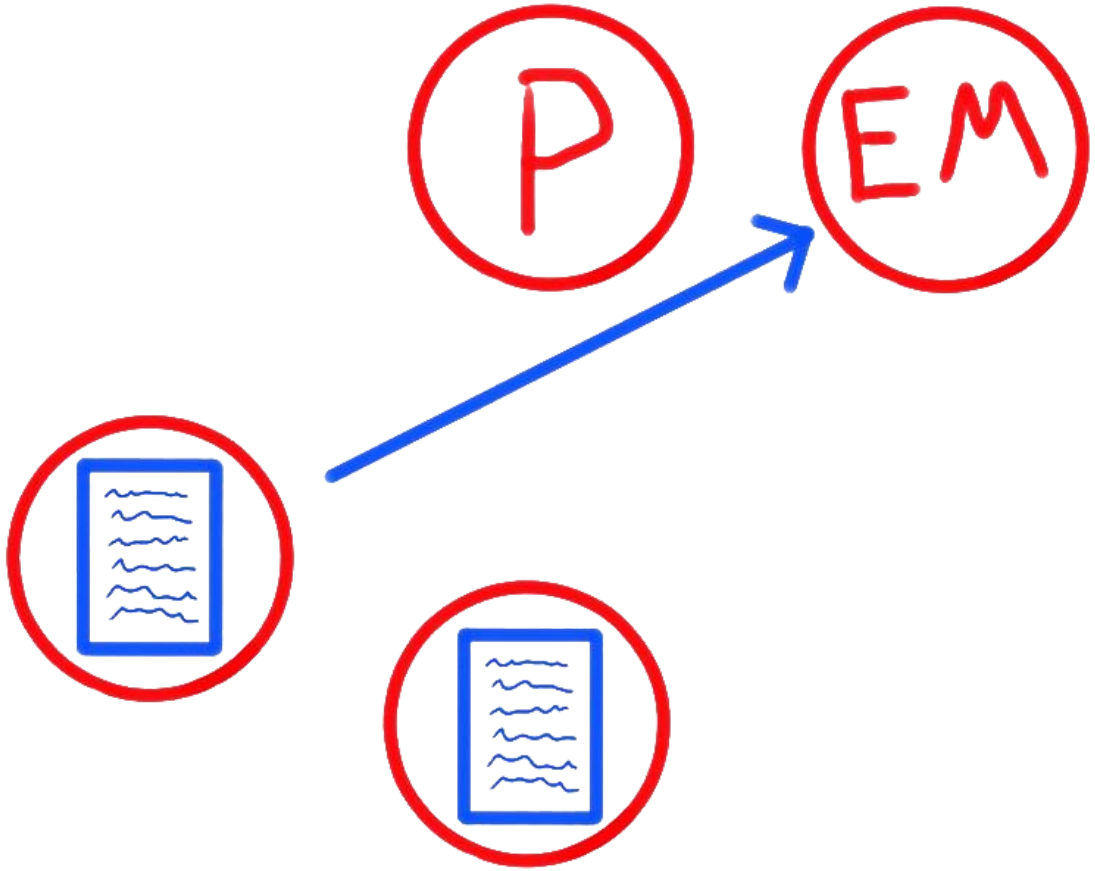
Asynchronous Decomposition

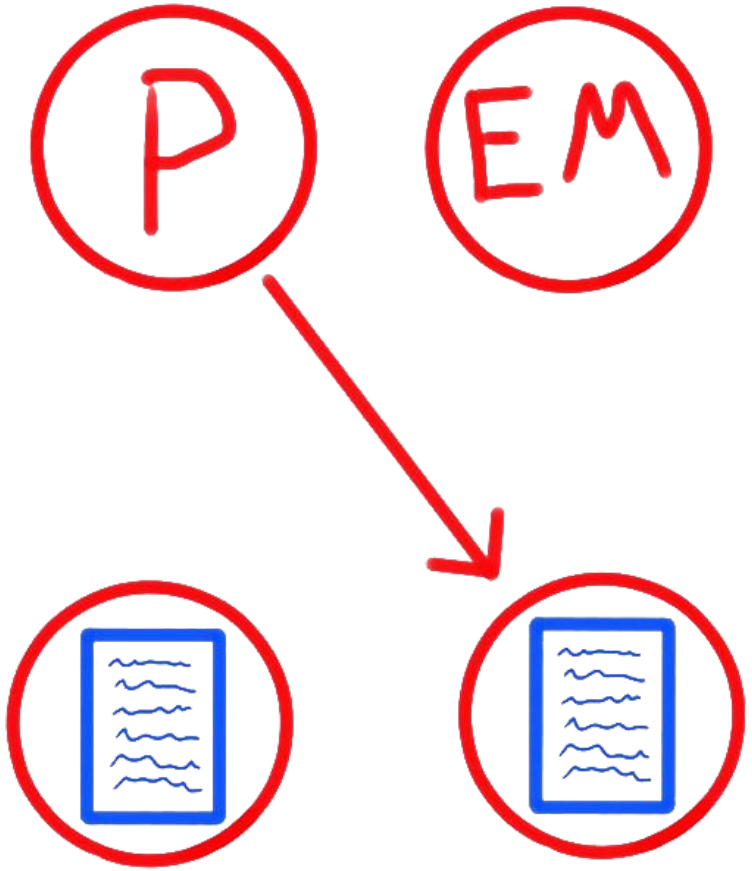
```
def run_module(mod) do
  EventManager.module_started(mod)

  mod
  |> get_tests()
  |> Enum.map(&run_test/1)

  EventManager.module_finished(mod)
end


def run_test(test) do
  EventManager.test_started(test)
  result = run(test)
  EventManager.test_finished(result)
end
```

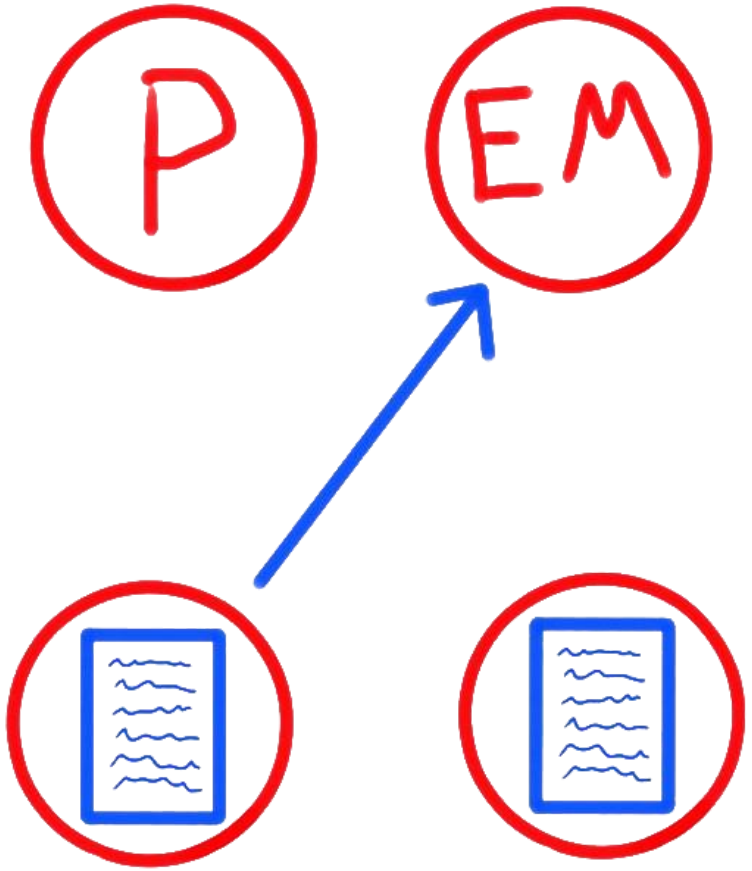

P
EM

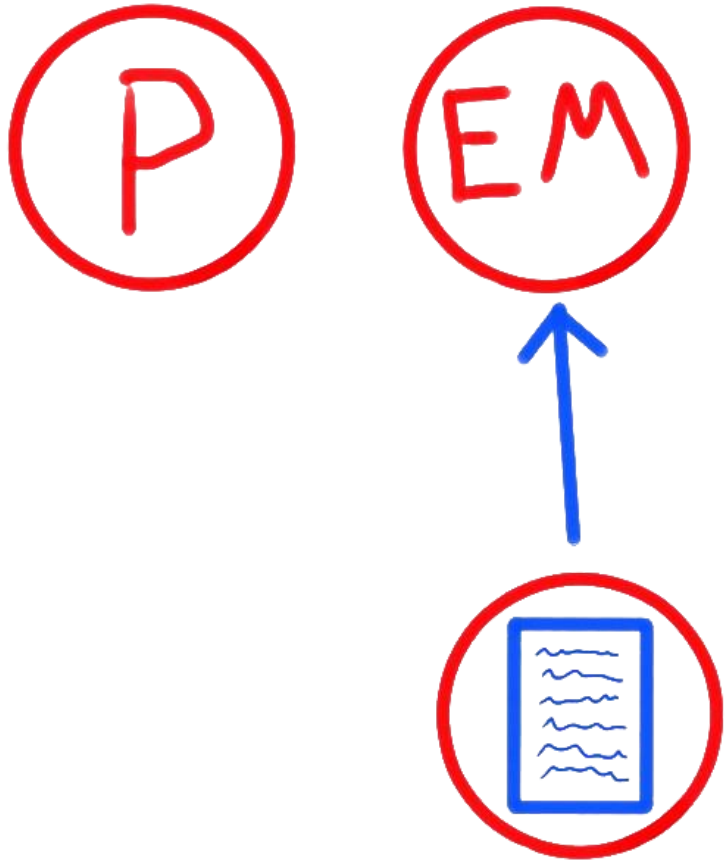

P
EM

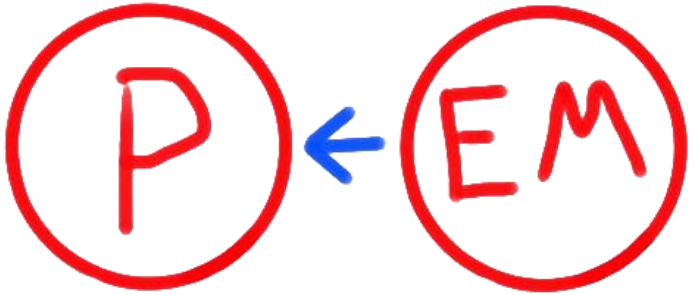

P
EM


P
EM


P
EM


P
EM


P
EM


P
EM

Asynchronous Decomposition

- Lazily evaluated input of unknown size

Asynchronous Decomposition

- Lazily evaluated input of unknown size
- Communication with parent allowed

Asynchronous Decomposition

- Lazily evaluated input of unknown size
- Communication with parent allowed
- No communication between siblings

Asynchronous Decomposition

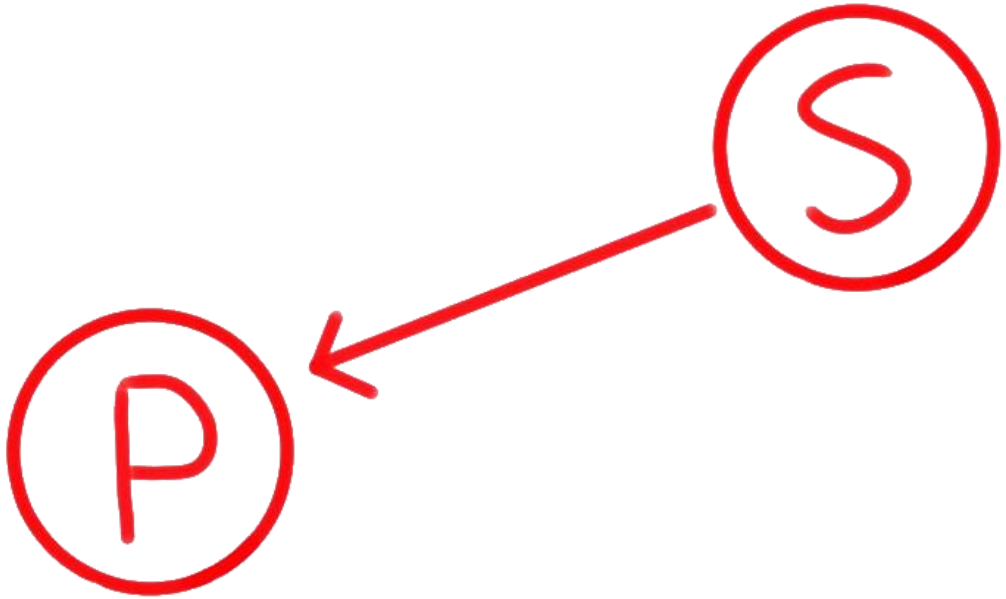
- Lazily evaluated input of unknown size
- Communication with parent allowed
- No communication between siblings
- No shared dependencies between siblings

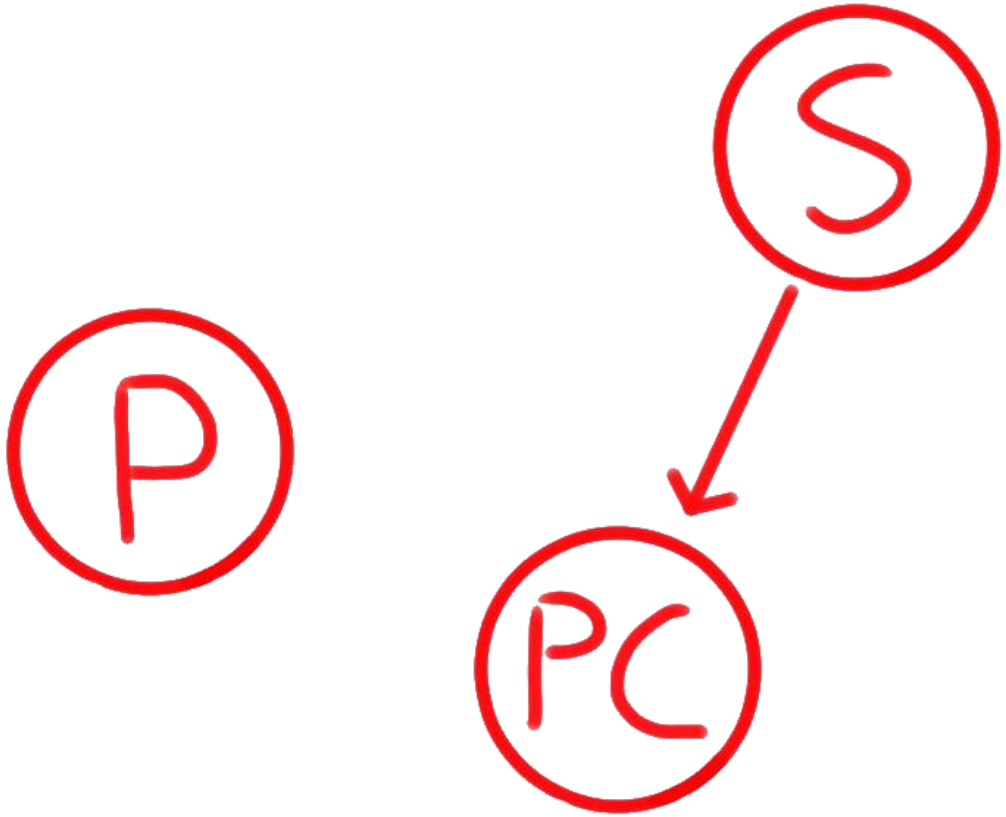
ExUnit As Reduction

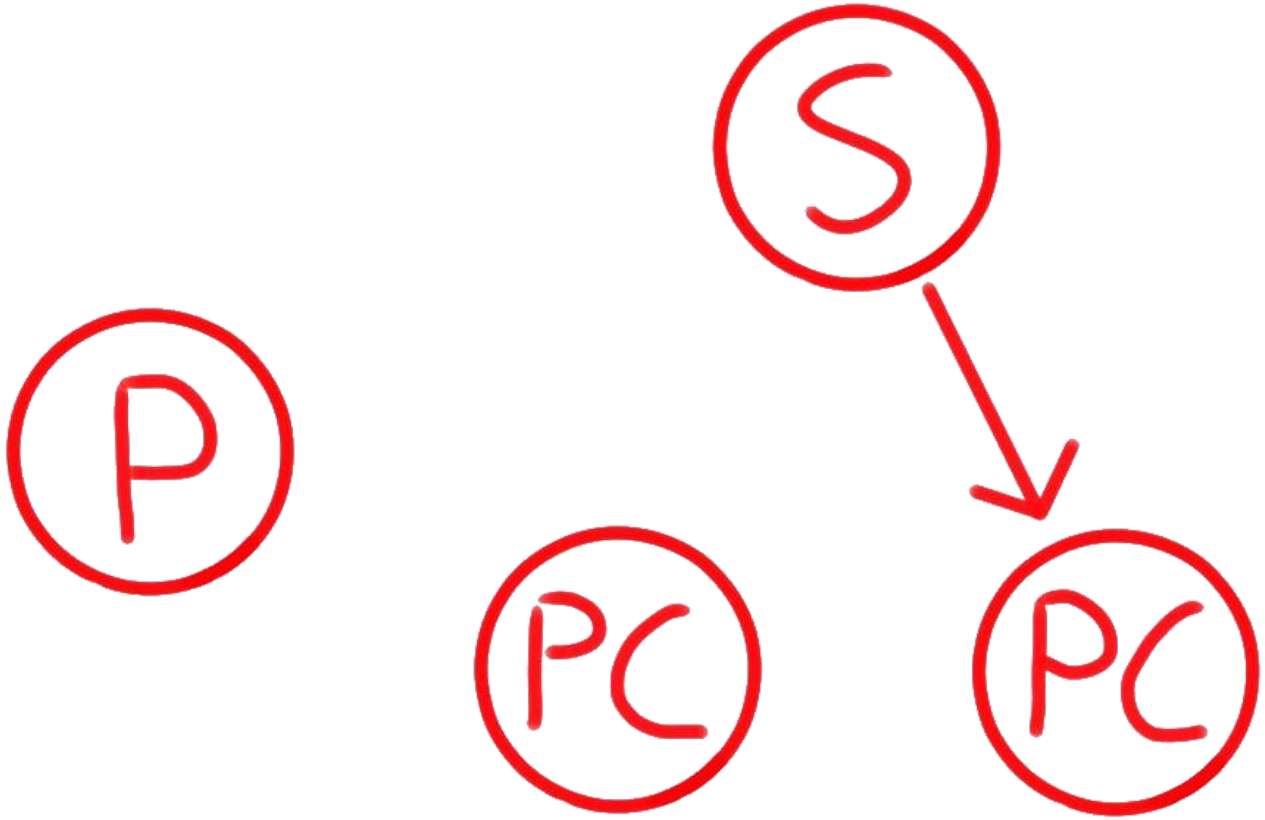
```
def run() do
  {sync_modules, async_modules} = load_modules()

  results =
    Parallel.map(async_modules, &run_module/1) ++
      Enum.map(sync_modules, &run_module/1)

  format_results(results)
end
```

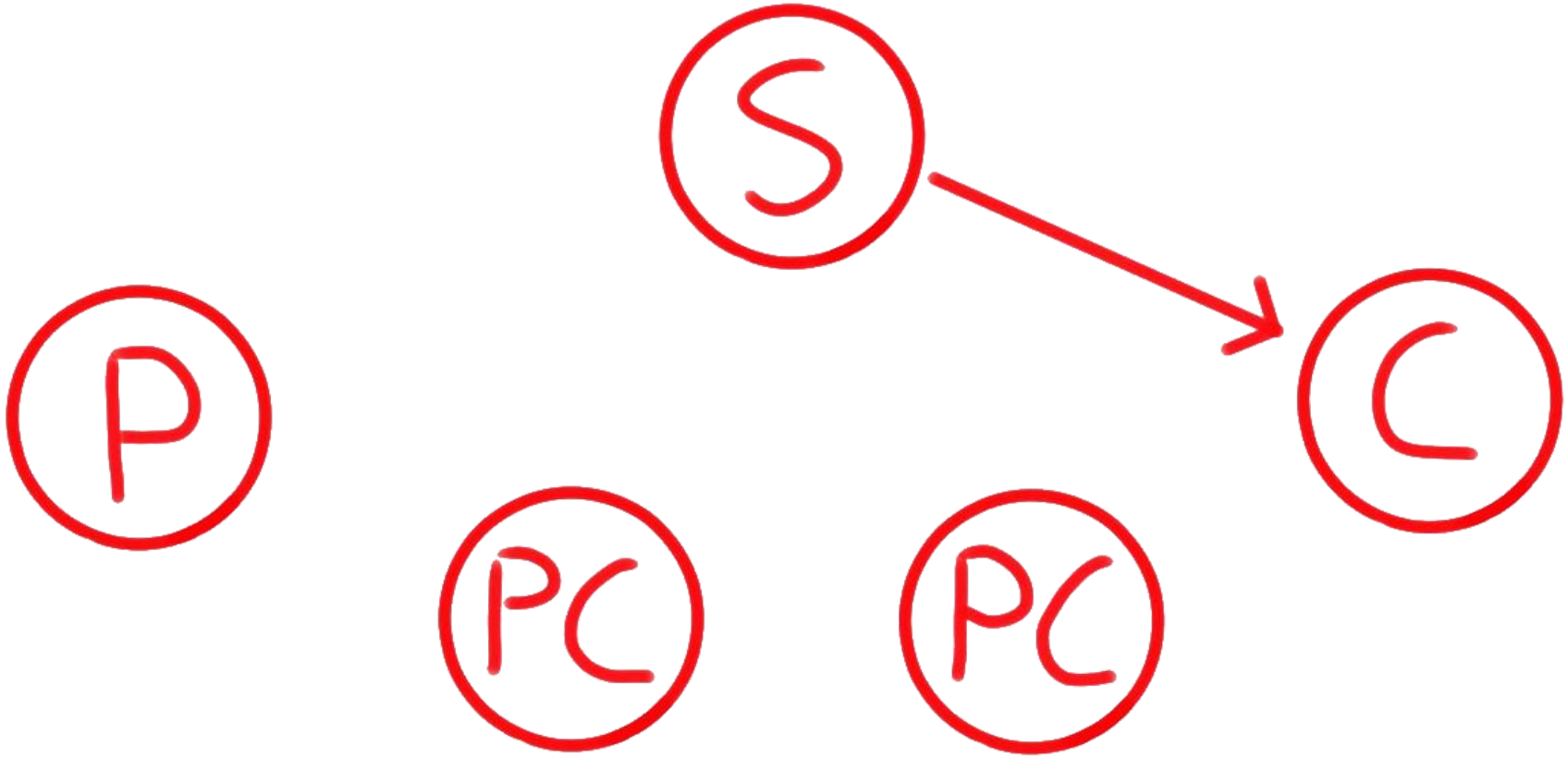
Pipeline Processing

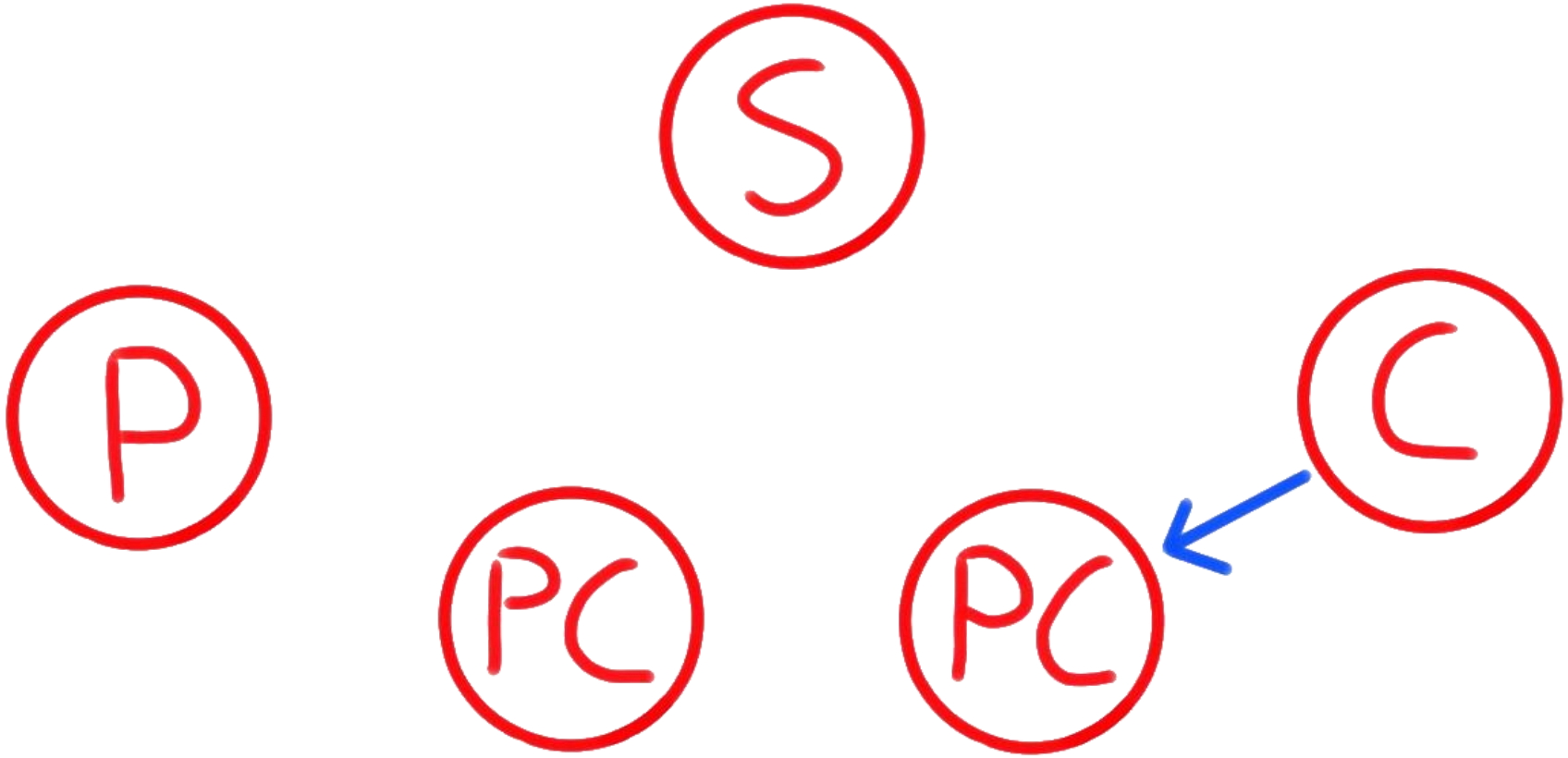

S

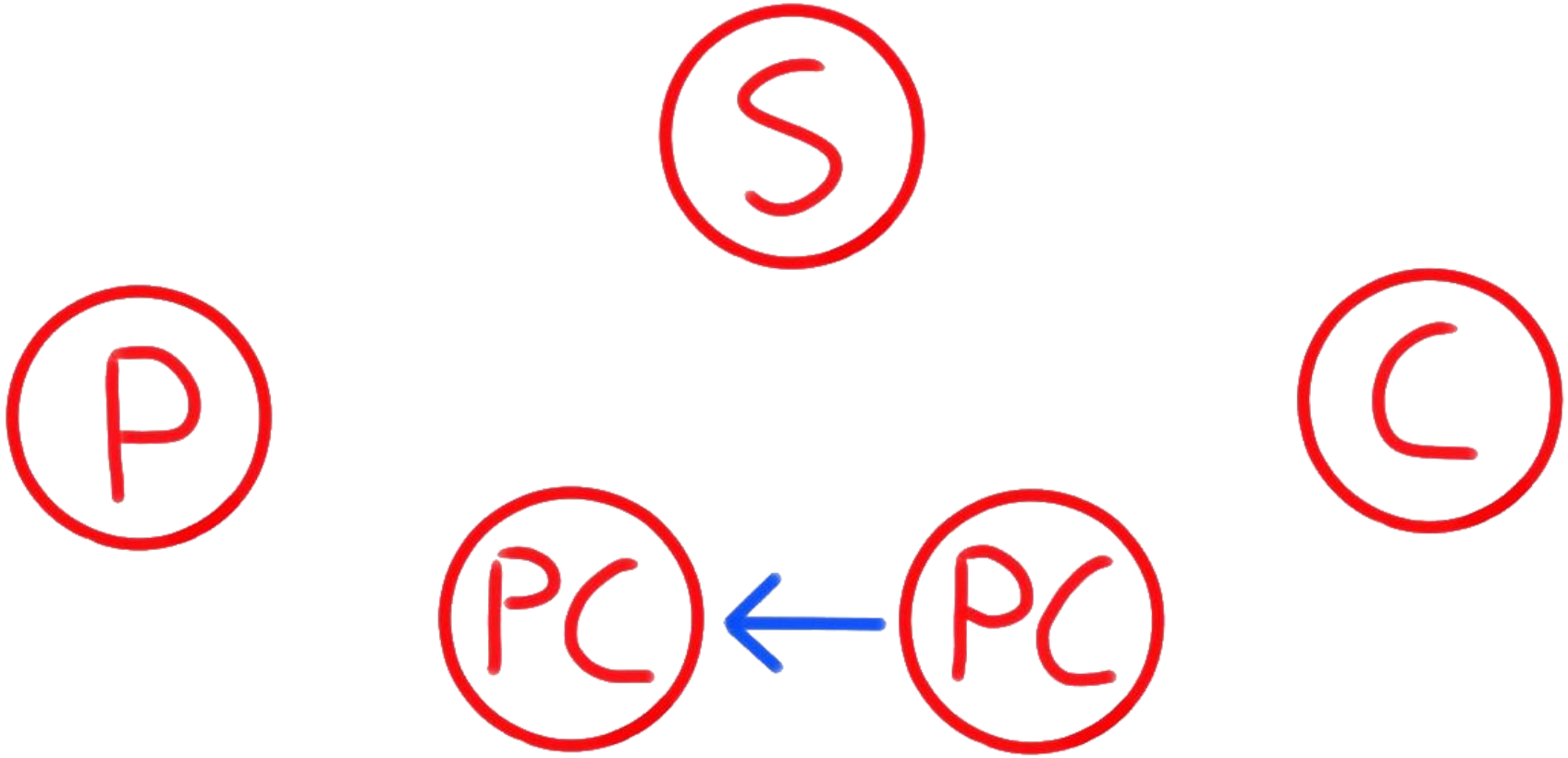

S
P

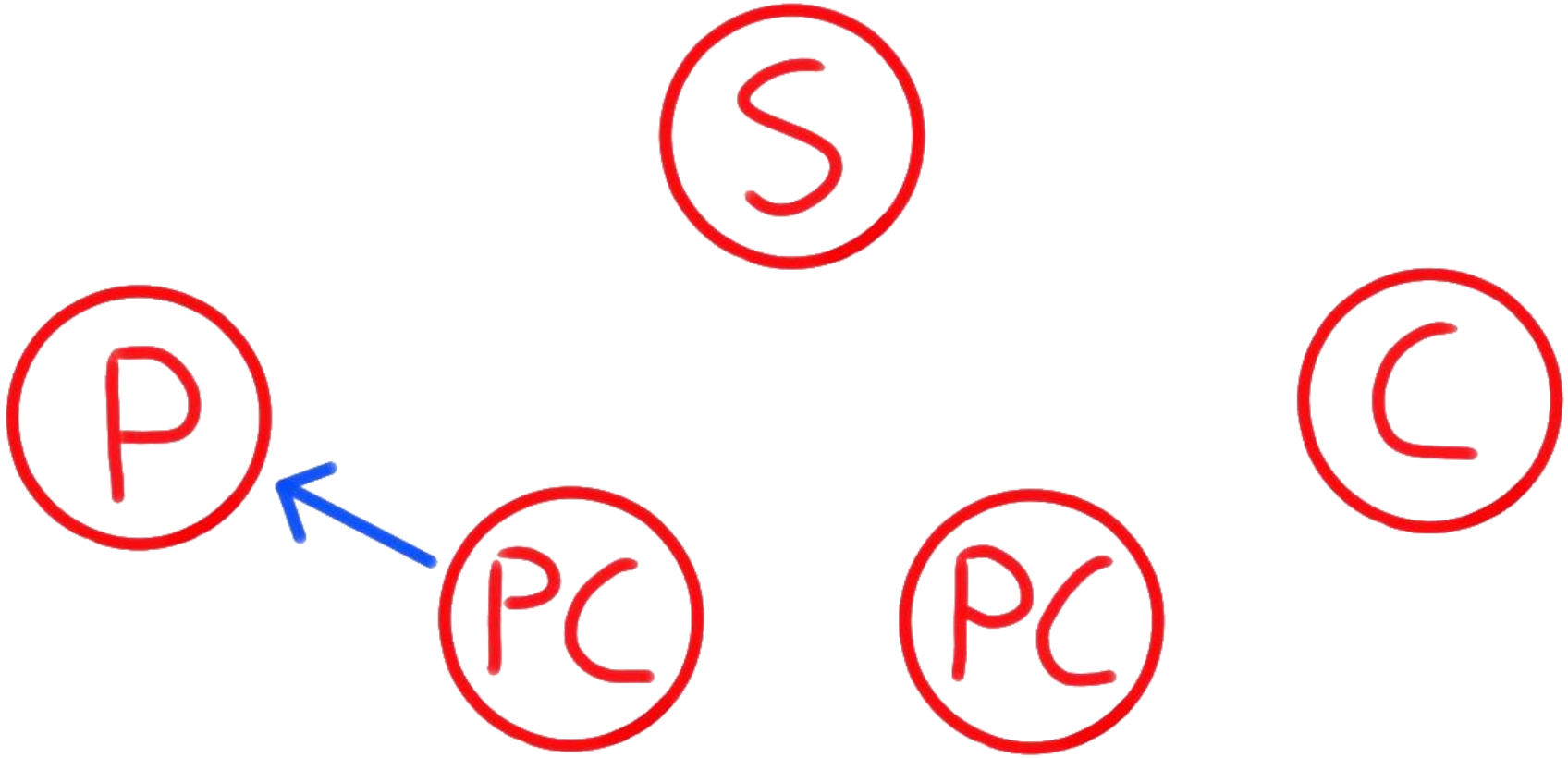

S
P
PC

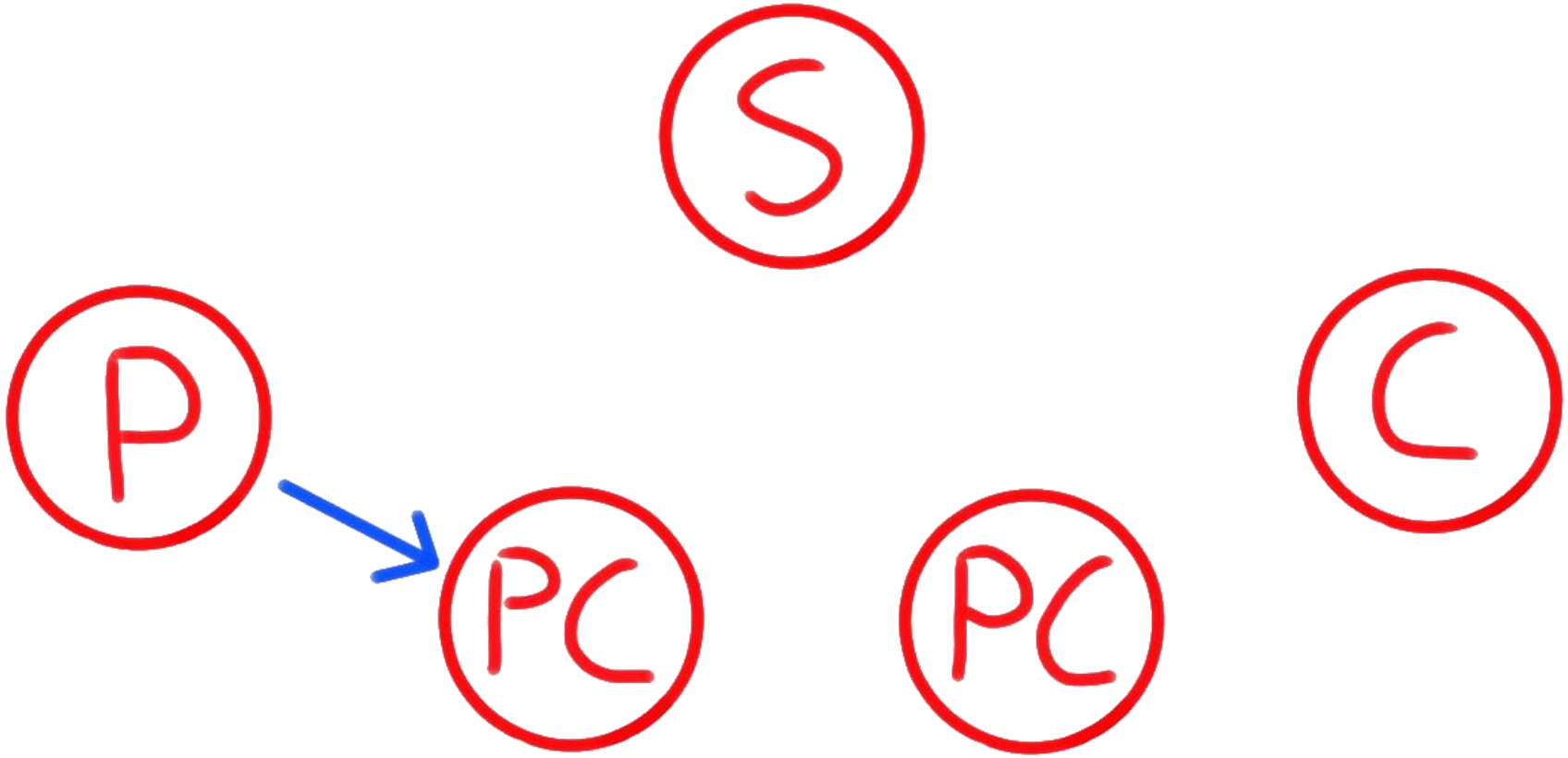

S
P
PC
PC

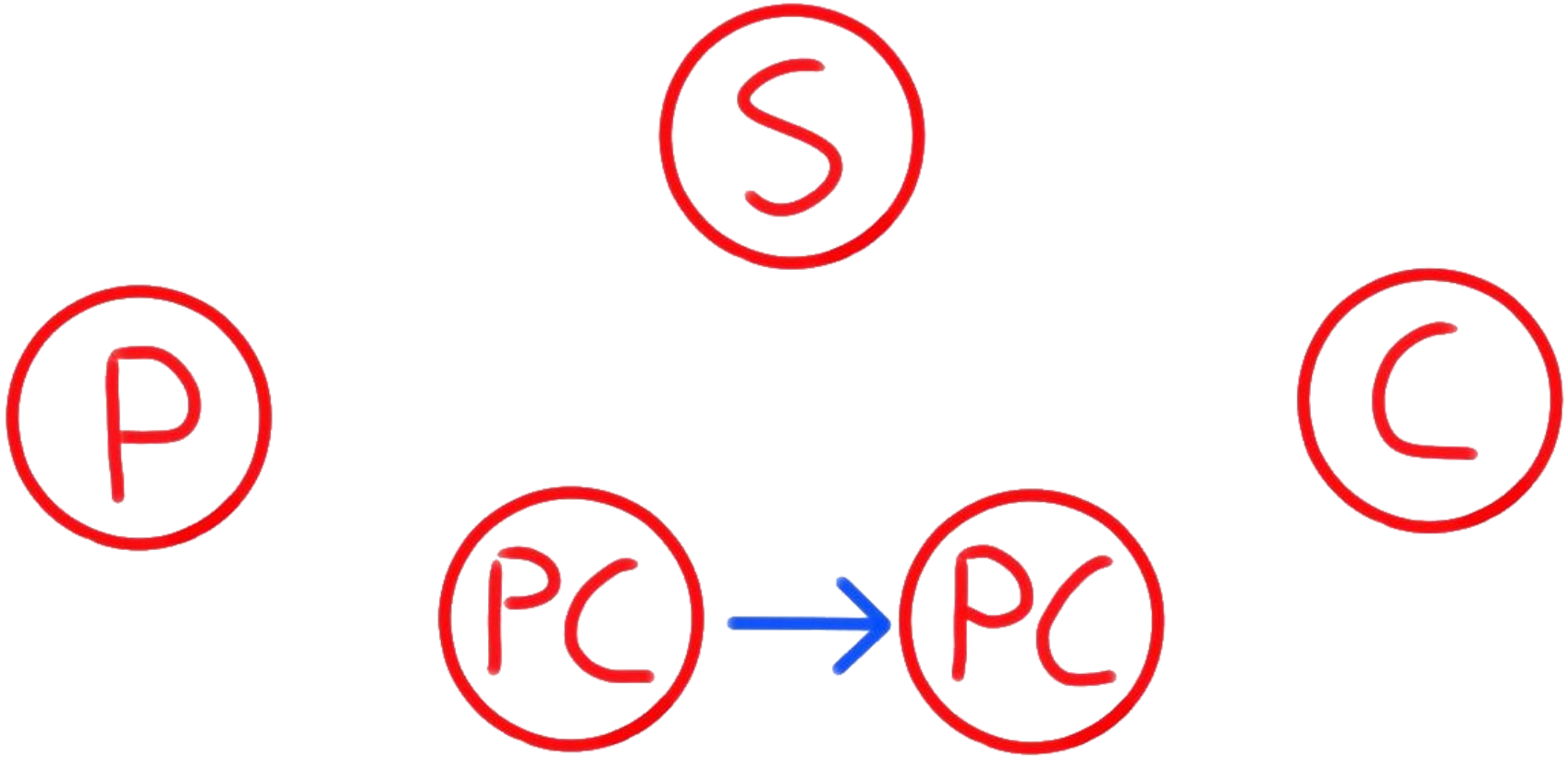

S
P
C
PC
PC

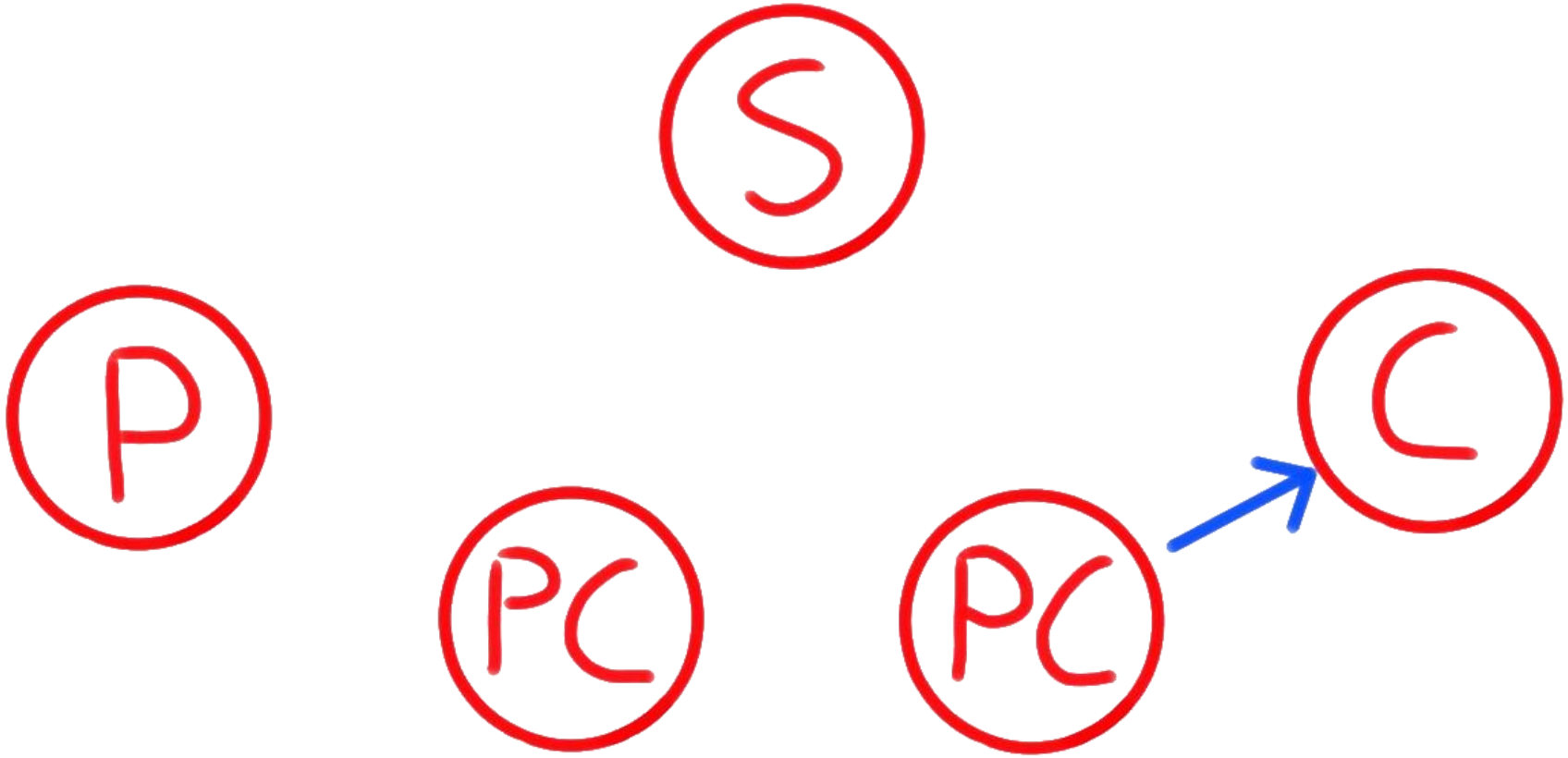

S
P
C
PC
PC


S
P
C
PC
PC


S
P
C
PC
PC


S
P
C
PC
PC


S
P
C
PC
PC


S
P
PC
PC
C

Pipeline processing

- Lazily evaluated collection of unknown size

Pipeline processing

- Lazily evaluated collection of unknown size
- Communication with parent allowed (rare)

Pipeline processing

- Lazily evaluated collection of unknown size
- Communication with parent allowed (rare)
- Communication between sibilings allowed

Pipeline processing

- Lazily evaluated collection of unknown size
- Communication with parent allowed (rare)
- Communication between sibilings allowed
- Shared dependencies allowed (but require management)

Summary

Eager	No	No	No

Summary

Eager	No	No	No
Eager	Yes	No	No

Summary

Eager	No	No	No
Eager	Yes	No	No
Lazy	Yes	No	No

Summary

Eager	No	No	No
Eager	Yes	No	No
Lazy	Yes	No	No
Lazy	Yes	Yes	Yes


Sketch

Thank you!