# PARALLELISM IN ERLANG

## HOW IT'S DONE AND HOW TO USE IT

PATRIK NYBLOM, ERICSSON AB
pan@erlang.org

# ERLANG/OTP AND ME...

- Erlang/OTP - department at Ericsson AB in Stockholm
- Developed the language and support libraries since 1996
- Open source since 1998, but still maintained at Ericsson
- SMP support since 2006
- I started 1998, and has since worked with...
  - VxWorks and Windows ports
  - ETS
  - SMP support
  - Distribution and network communication
  - Virtual machine core
  - Garbage collector
  - Tracing
  - Unicode, Regular expressions, Dtrace support, ...
  - Team leader of VM team for several years
  - Lecturing at Stockholm University in parallel programming

# PARALLELISM IN ERLANG

- Erlang is an actor language with a functional sequential part
- Actors (processes) execute independently
  - Concurrency built in
- Implicit parallelism
  - With the SMP support concurrency became parallelism
    - The more concurrency, the more implicit parallelism
    - Requires suitable design choices
- Explicit parallelism
  - The implicit parallelism can be utilized to create truly parallel programs in a more classical way
  - Dividing algorithms into autonomous parts that can run in parallel
  - Create at least an actor (Erlang process) for each core in the system
    - The VM will spread working processes over the cores

# IMPLICIT PARALLELISM IN ERLANG

- “Normal” concurrent Erlang processes (actors) are automatically spread over the available CPU cores so that they can execute in parallel
- The programmer basically writes the programs according to the paradigm, without ever thinking about threads, cores shared resources etc
  - Still requires that the program really has potential for parallelism
  - If nothing executed concurrently (all synchronous behavior), nothing will happen in parallel
  - Most communication programs will gain automatically
  - Programs that use purely synchronous interfaces between actors will gain next to nothing at all
    - Will need rethinking

# IMPLICIT PARALLELISM IMPLEMENTATION

- The VM schedules the execution of actors with one scheduler per core in the system.
- As many actors as there are cores will execute in parallel at any time
- Elaborate (and unique) algorithms for distributing actors over the scheduling queues
  - Put the job in the “shortest” queue
  - Rebalance periodically
  - Migrate and steal jobs when necessary (but only then)
  - Try to keep processes running in the same core
- Communication (message passing) between the actors (and possibly between cores) are handled by the VM
  - The programmer does not need to know anything about the message queue implementation
- The VM takes responsibility for all communication between schedulers
  - New code
  - Shared resources like ETS tables

# IMPLICIT PARALLELISM, BAD EXAMPLE

```
start(PortNumber) ->
   {ok, ListenSocket} =
     gen_tcp:listen(PortNumber,[{active,false},{backlog,100}]),
   server_loop(ListenSocket).
server_loop(ListenSocket) ->
   case gen_tcp:accept(ListenSocket) of
     {ok, Sock} ->
       handle_connection(Sock),
       gen_tcp:close(Sock),
       server_loop(ListenSocket);
     Other ->
       handle_error(Other)
   end.
 handle_connection(Sock) ->
  %% gen_tcp:recv -> gen_tcp:send until done
```

# IMPLICIT PARALLELISM, BETTER EXAMPLE

```
server_loop(ListenSocket) ->
  case gen_tcp:accept(ListenSocket) of
    {ok, Sock} ->
      Ref = make_ref(),
      Pid = spawn(?MODULE,handle_connection,[Sock,Ref]),
      gen_tcp:controlling_process(Sock, Pid),
      Pid ! Ref,
      server_loop(ListenSocket);
    Other ->
      handle_error(Other)
  end.
handle_connection(Sock, Ref) ->
  receive Ref -> conn_loop(Sock) end.
conn_loop(Sock) ->
 %% gen_tcp:recv -> gen_tcp:send until done, then close.
```

# OR WITH A NUMBER OF PRE-SPAWNED WORKERS

```
start(PortNumber) ->
   {ok, ListenSocket} =
     gen_tcp:listen(PortNumber,[{active,false},{backlog,100}]),
   Plist =
     [spawn(?MODULE,server_loop,[ListenSocket]) ||
       _ <- lists:seq(1,erlang:system_info(schedulers_online)*
                         ?SOME_ FACTOR)],
   wait_for_processes(Plist).
server_loop(ListenSocket) ->
   case gen_tcp:accept(ListenSocket) of
     {ok, Sock} ->
       handle_connection(Sock),
       gen_tcp:close(Sock),
       server_loop(ListenSocket);
     Other ->
       handle_error(Other)
   end.
handle_connection(Sock) ->
   %% gen_tcp:recv -> gen_tcp:send until done
```

# EXPLICIT PARALLELIZATION

- If one single actor has to much work to do compared to other actors in the system, the work will not be equally distributed among cores
  - Usually not a big problem in large communication systems (telecom, web servers etc)
  - A really big problem in e.g. AI algorithms, compilers etc where the program is sequential
- The algorithm needs to be divided into tasks that can be performed in parallel
  - Sometimes it's easy - the algorithm is embarrassingly parallel
    - Typically some matrix operations, GPU's has many cores
    - Usually data parallel
    - Still requires care! Is it really embarrassingly parallel to the computer?
  - Usually it's harder
    - Task dependencies
    - Shared resources...

# SIMPLE EXAMPLE OF TASK DEPENDENCIES

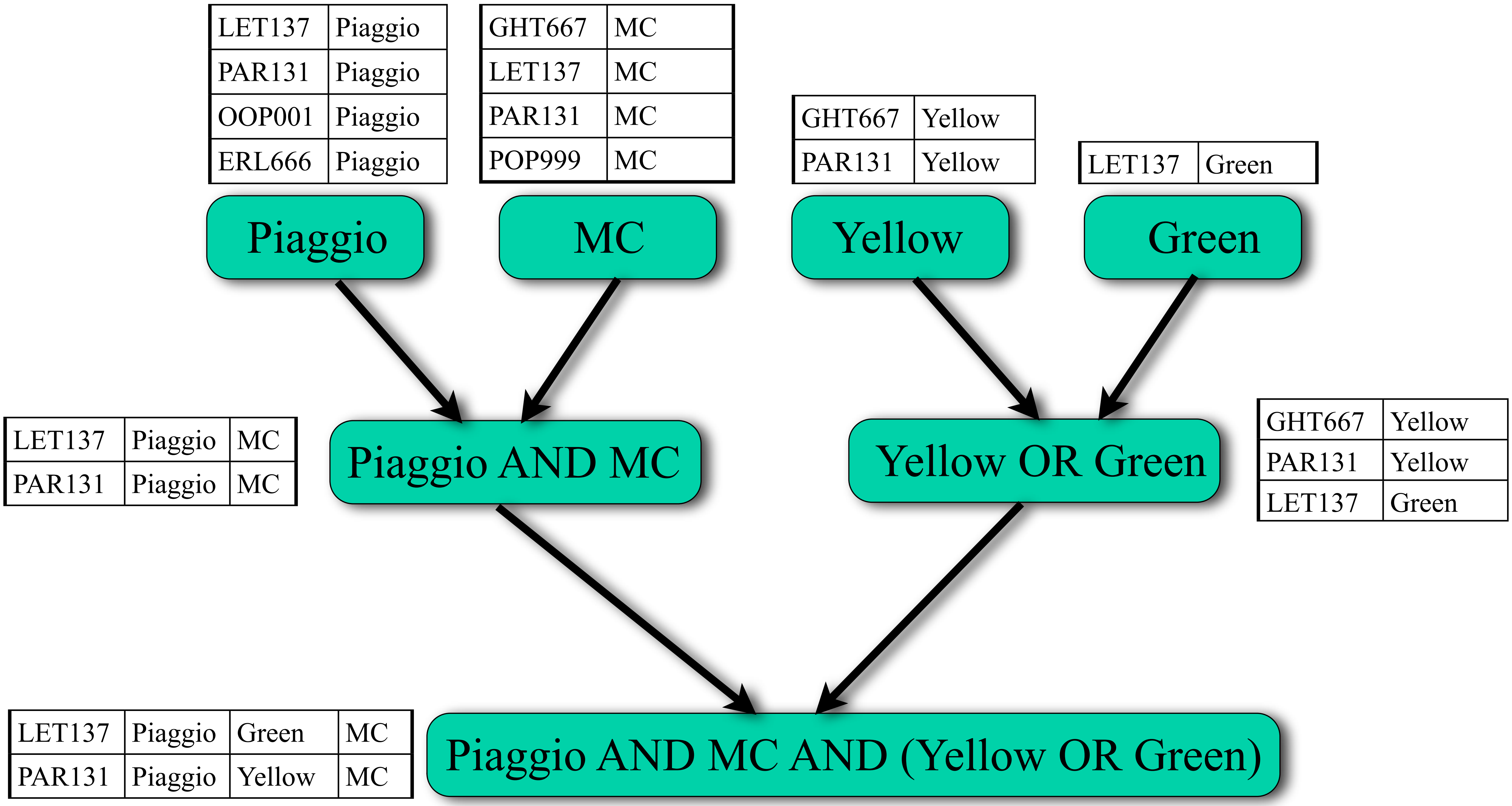
- A really simple database table

| RegNo | Brand | Model | Year | Colour | Type |
|---|---|---|---|---|---|
| LKM678 | Saab | 9-5 | 2006 | Blue | Car |
| GHT667 | Triumph | Thruxton | 2008 | Yellow | MC |
| LET137 | Piaggio | Vespa | 1999 | Green | MC |
| PAR131 | Piaggio | Vespa | 1967 | Yellow | MC |
| OOP001 | Piaggio | Vespa | 1969 | Red | Moped |
| ERL666 | Piaggio | Vespa | 1968 | Red | Moped |
| POP999 | Lambretta | LI150 | 1960 | Blue | MC |

- Let's say we want the reg. numbers of all the Piaggio MC's that are either Yellow or Green:

```
SELECT RegNo FROM Vehicles WHERE
   Brand="Piaggio" AND Type="MC" AND
   (Colour="Yellow" OR Colour="Green");
```

# LETS DIVIDE THE QUERY INTO TASKS




| | |
|---|---|
| LET137 | Piaggio |
| PAR131 | Piaggio |
| OOP001 | Piaggio |
| ERL666 | Piaggio |

| | |
|---|---|
| GHT667 | MC |
| LET137 | MC |
| PAR131 | MC |
| POP999 | MC |

| | |
|---|---|
| GHT667 | Yellow |
| PAR131 | Yellow |

| | |
|---|---|
| LET137 | Green |

| | | |
|---|---|---|
| LET137 | Piaggio | MC |
| PAR131 | Piaggio | MC |

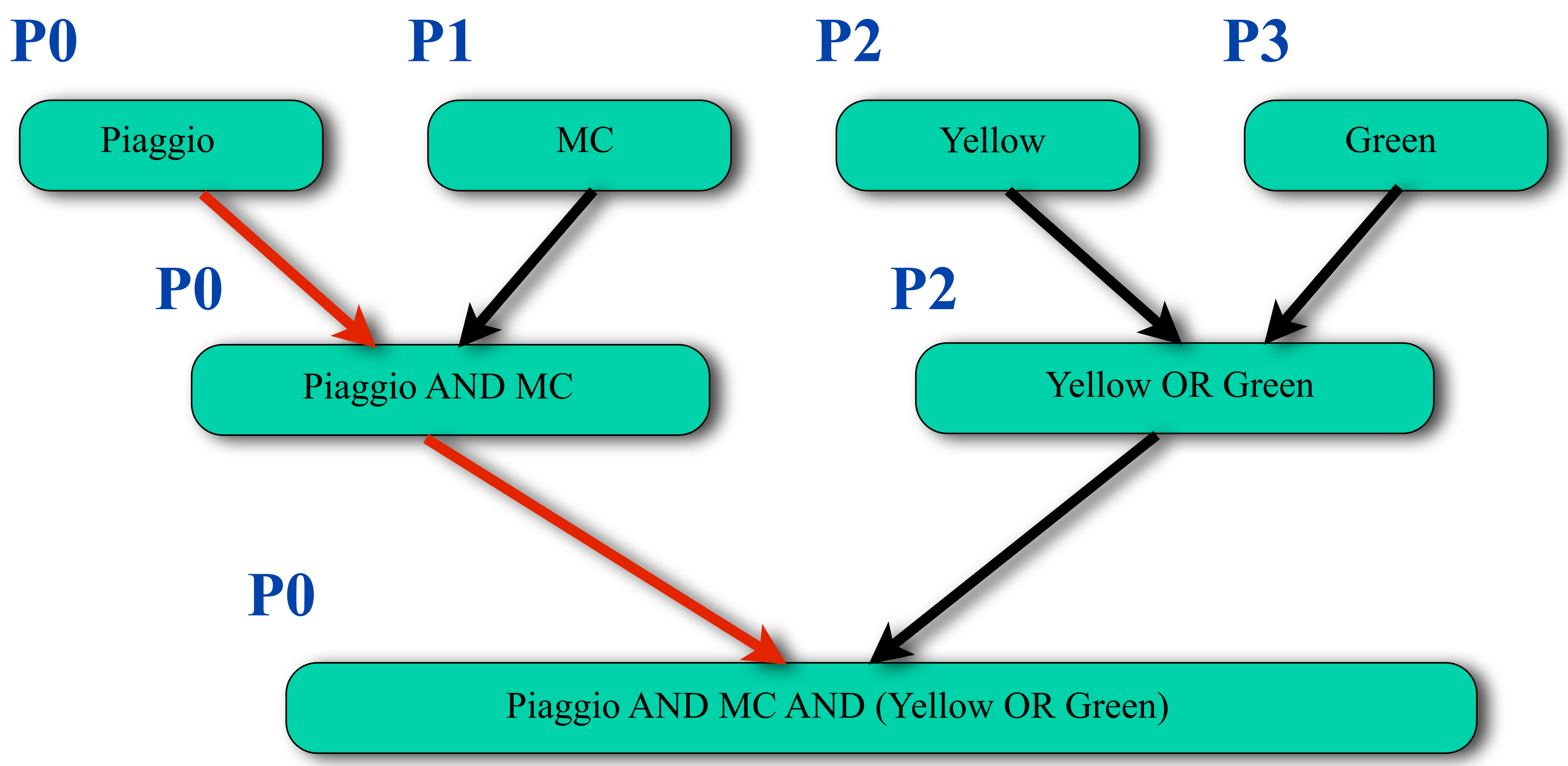
| | |
|---|---|
| GHT667 | Yellow |
| PAR131 | Yellow |
| LET137 | Green |

| | | | |
|---|---|---|---|
| LET137 | Piaggio | Green | MC |
| PAR131 | Piaggio | Yellow | MC |



onsdag den 2 maj 2012

# YOU COULD THEN SPREAD THE TASKS OVER UP TO FOUR CORES...



# DECOMPOSING OTHER PROBLEMS THAN THE SIMPLEST...

- Tasks may be unbalanced, size of tasks may be different
- Average sizes of tasks may vary depending on the problem
- Dependencies may come in different forms
  - Unbalanced as an ad hoc query to a database
  - Balanced like a merge-sort
    - Divide and conquer
    - Recursive decomposition
- Some tasks may even be wasted
  - Exploratory decomposition
  - Typically in search algorithms
- The number and sizes of tasks may be known from the beginning
  - Static vs dynamic task creation
- Communication between tasks means synchronization points and possible congestion - should be kept at a minimum
- The memory consumption and placement are very important characteristics that are often not fully understood

# MAPPING THE TASKS TO PROCESSORS

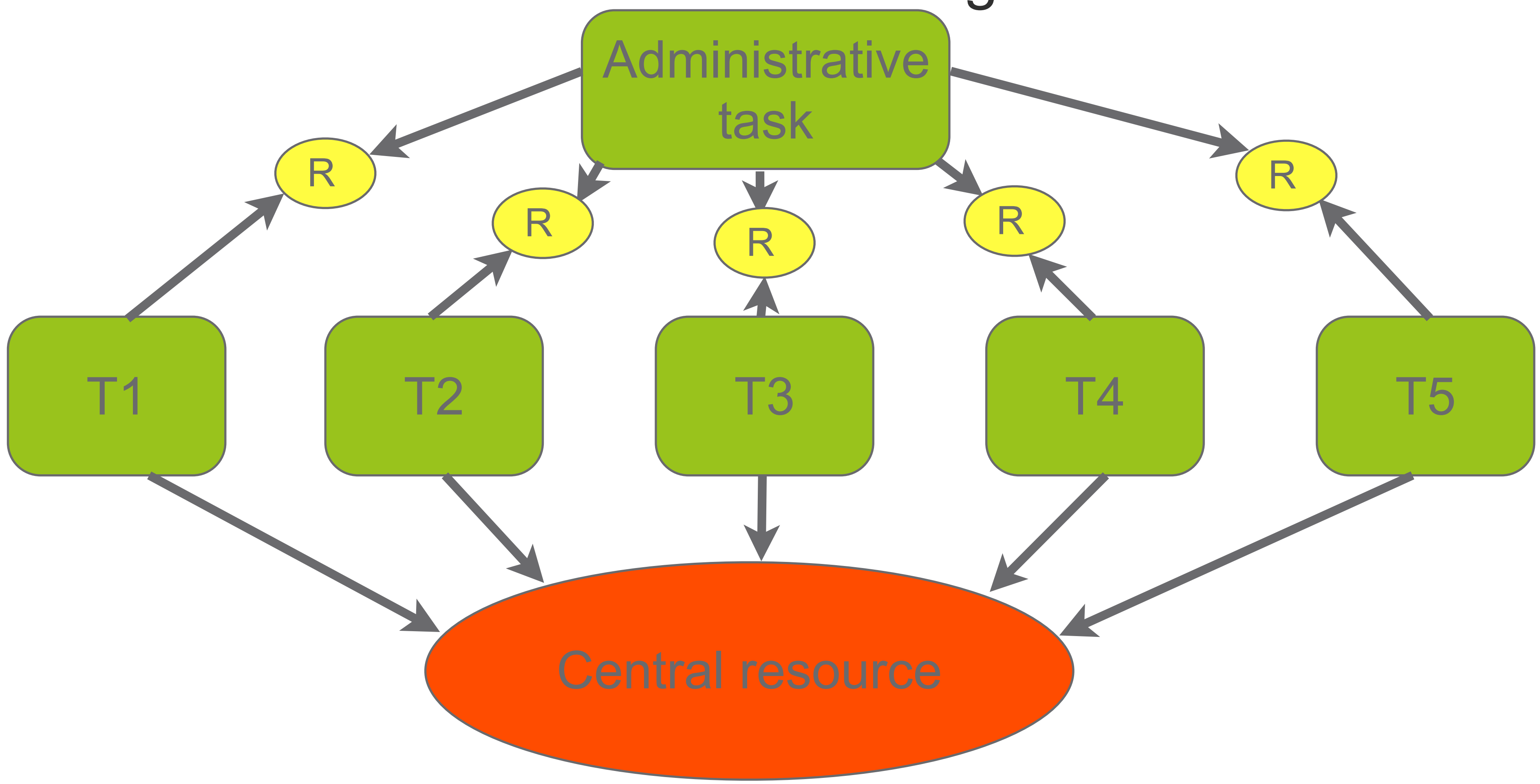
- In a static decomposition, one might create one actor per core in the system and spread the tasks equally
- As soon as tasks are of indeterminable sizes or dynamically created, it get's trickier
  - In Erlang, a shortcut is to create more actors than cores (schedulers), The scheduling algorithm will dynamically spread the work over the cores
    - Sometimes you can simply create one process per task
    - For smaller tasks, you can create actors up to a defined limit and let them work on the tasks available
  - In other languages (non-actor), a worker pool is the way to achieve the effect of actors
  - A lot of tricks for worker pools can be found in communicating applications, where resources are dynamically allocated to handle certain types of load

# HOW IS THIS DONE INSIDE THE MACHINE?

- The job an actor does during one scheduling equals the tasks
- The schedulers are the worker pool
  - Avoid communication between the workers (i.e. the schedulers). They...
    - Have their own queues
    - Have their own memory allocators
    - Tend to schedule jobs for batch processing at given intervals
  - Lock-free queues are used for much of the communication
    - Lock free algorithms are seldom wait-free
    - Hard to invent, hard to prove
    - Understanding the processor architectures in detail is necessary
      - Use of memory barriers is essential...and hard...
      - Forget the concept of “now”, what you have is more of if - then relationships (implications)
  - Atomic operations is a special kind of small lock-free algorithms
    - Built into modern architectures
    - May be more costly than you think (e.g. atomic exchange may requires a lot of communication)

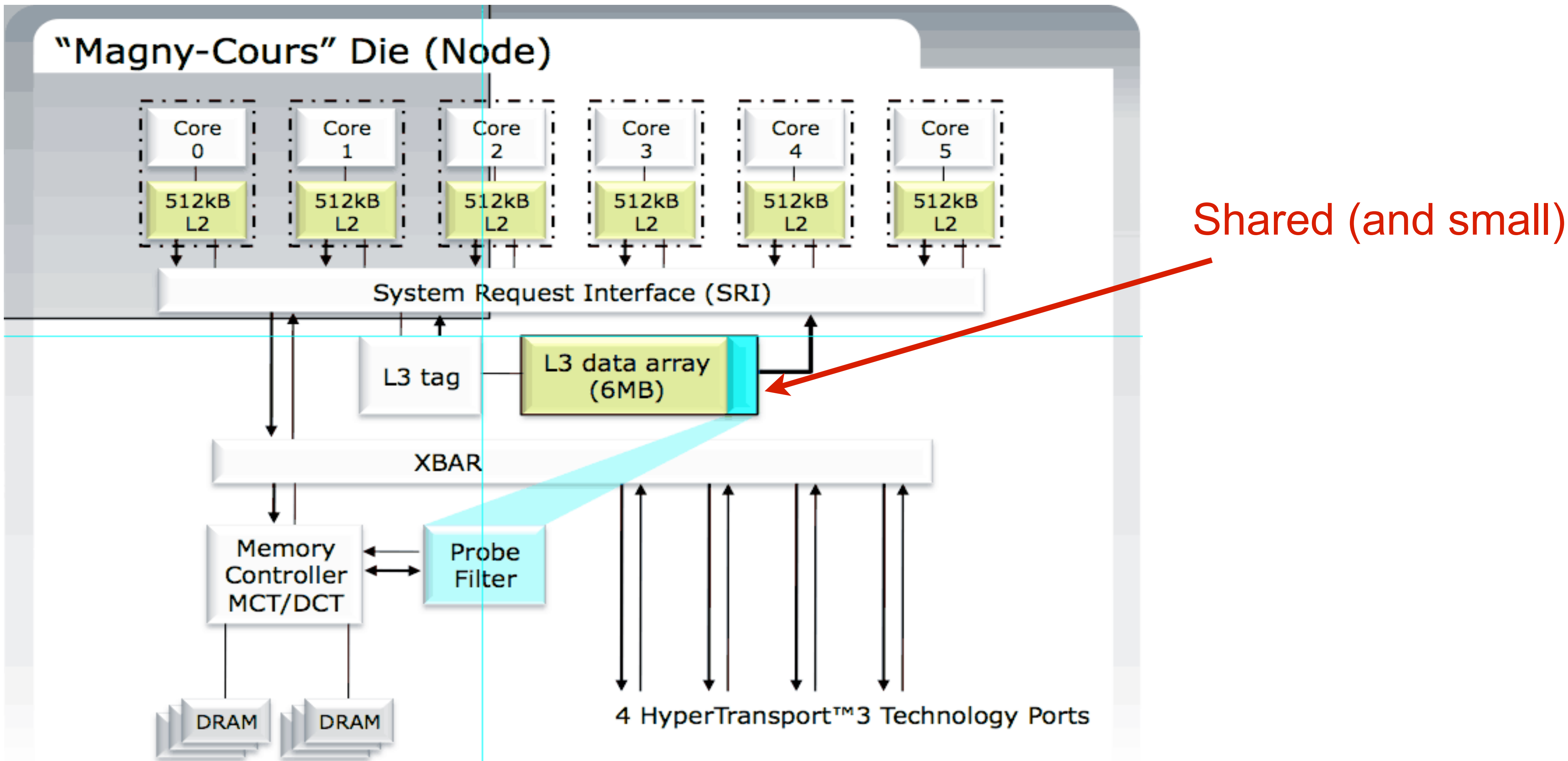
# HARDWARE DEVELOPMENT DRIVES VM DEVELOPMENT

- Multicore to many-core
  - Global locks get more or less impossible to use: congestion eats up the gain of additional cores
    - Use lock free algorithms
    - Add management tasks with multiple message queues collecting information from the workers to handle global data



# HARDWARE DEVELOPMENT DRIVES VM DEVELOPMENT (CONTINUED)

- Modern architectures share resources between cores that you might not expect
  - Example: AMD magny-cours - one of the two NUMA nodes in a processor

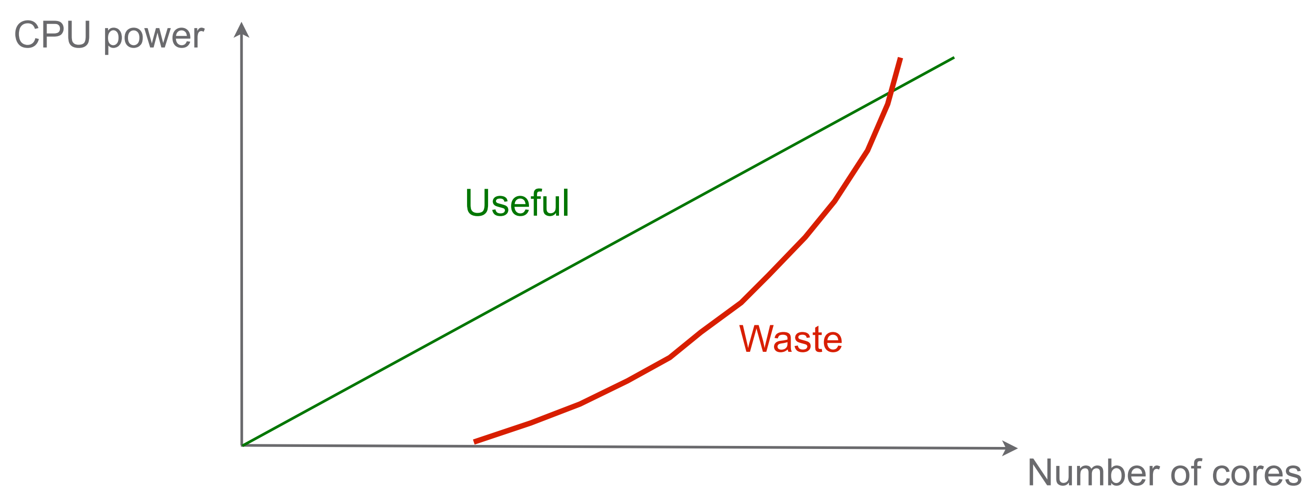


# CACHES ARE IMPORTANT IN SO MANY WAYS…

- Sharing a cache line between cores lead to cache “ping-pong”
  - Try not to access shared memory frequently
- Having a large active working-set in one processor can give cache misses in another
  - Opteron not the only such architecture
- A constant tradeoff
  - Keep a schedulers memory separate
  - Keep the memory consumption at a minimum (often functional programs weak spot)

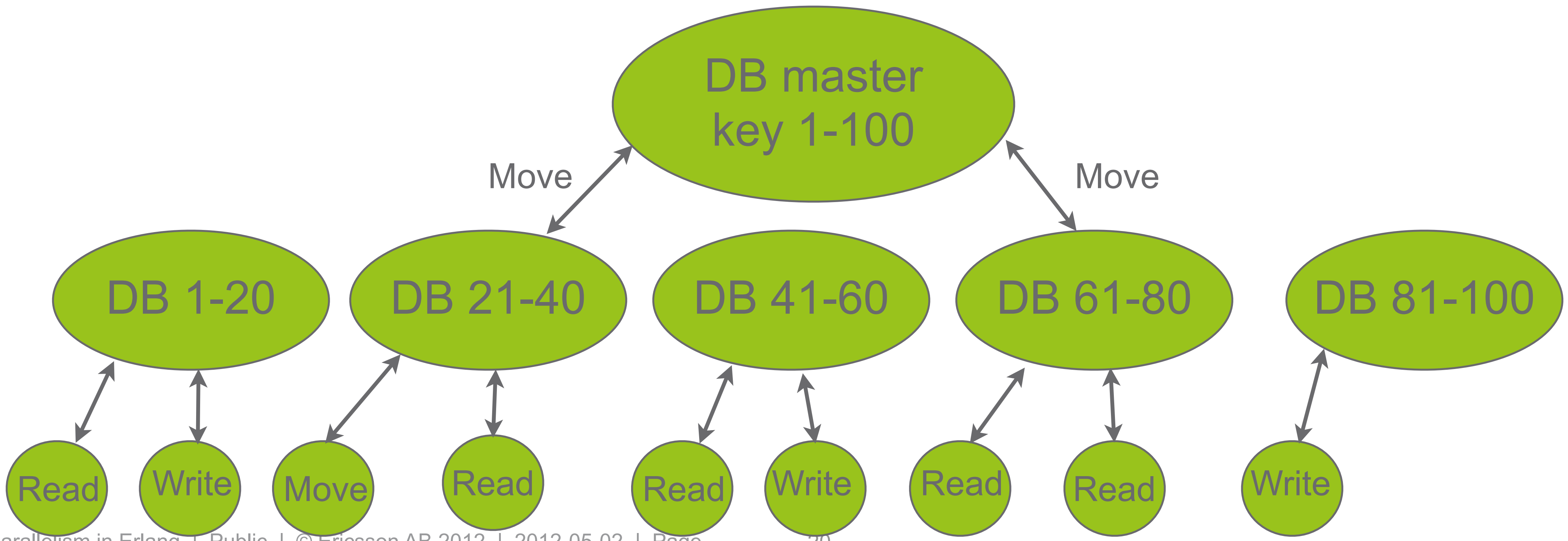
# TO SCALE OVER MORE AND MORE CORES…

› You should not have algorithms where the waste part increases with each new core
  - A well thought through strategy for each shared concept
  - Be aware of the whole system
› You should have algorithms that allow cores to be added
  - Algorithms divided into to big tasks can not utilise any number of cores



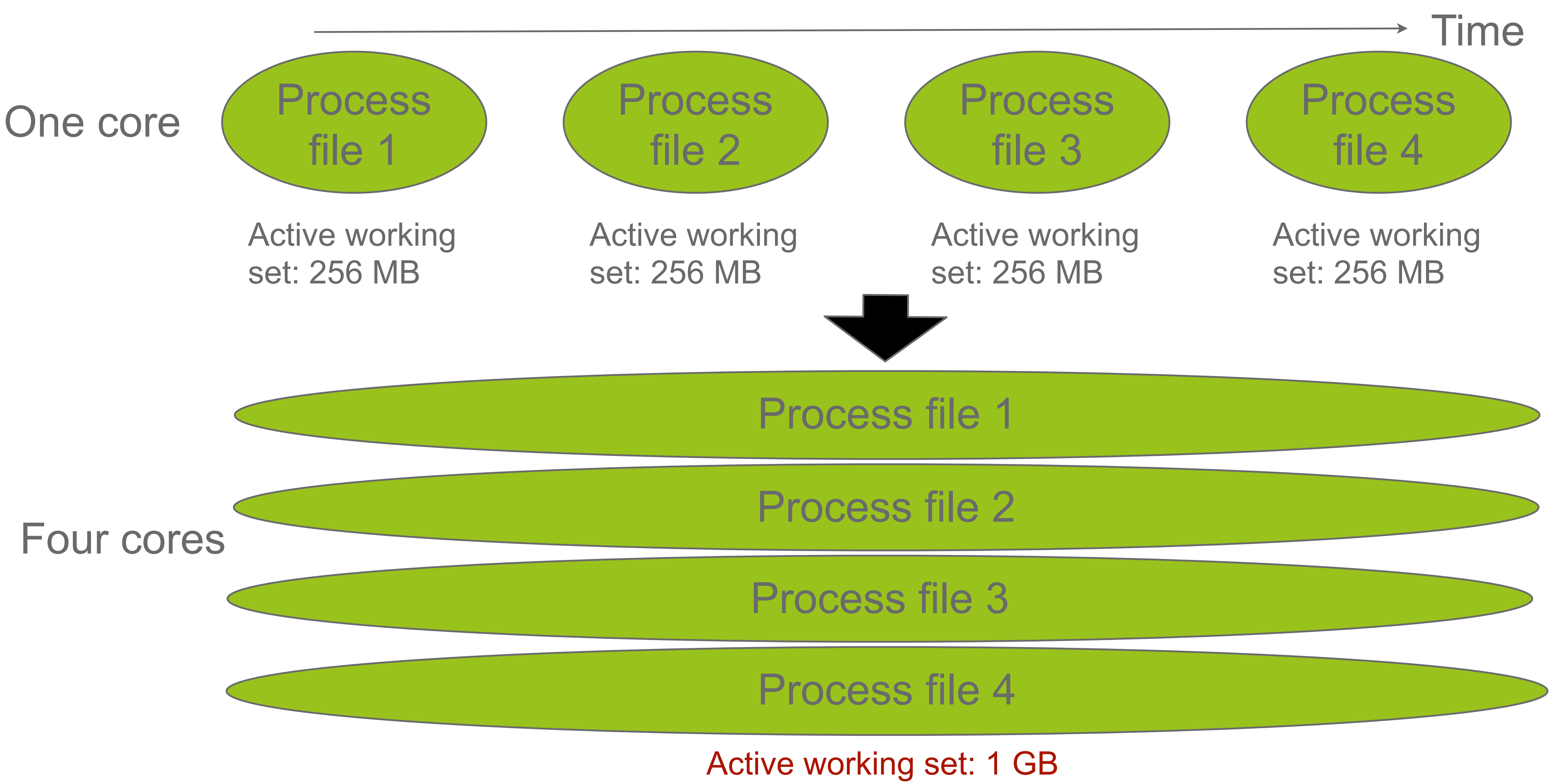
# MOVING BACK TO ERLANG…

- A shared resource is i.e. an ETS table or a central server
  - Don’t have central resources
  - If a central ETS table is needed, use read_concurrency and batch writes if possible
  - If a shared server is used, make sure it works as asynchronous as possible
    - Always answer a client at the earliest possible point
    - Carefully utilize asynchronous message passing
  - Build hierarchies of processes if possible
    - Workers to offload the server
    - Interface processes/caches



# THINK ABOUT THE CACHE AND MEMORY - EVEN IN ERLANG

› Don't fall into the "multiplying memory consumption trap"
  - Divide the active working set when parallelizing
  - A common way to parallelize is to do already independent tasks in parallel instead of sequentially:



# PARALLELIZING IN ERLANG...

- Often you don't need to (it's implicitly done)
- When you need to explicitly parallelize:
  - Algorithm decomposition into tasks is done in the same way as in any language
  - The functional language gives you power to implement complex algorithms
  - The actor model gives a lot of help when parallelizing the complex algorithm
  - The implementation of the VM helps you utilize the resources of the machine
  - ...but a bad decomposition of a problem is bad in any language
- There is still a computer down there you will have to relate to
  - It just behaves more friendly when you write in Erlang...

ERICSSON