Parallel Programming in Erlang using Skel

Chris Brown
University of St Andrews

cmb21@st-andrews.ac.uk

Outline

Basic course in Parallel Programming using Skeletons in Erlang

1. Introduction to “Skel”
2. Building patterns in Skel
   1. Seq
   2. Pipeline
   3. Farm
   4. Combining
3. Introducing granularity

By the end, you will be able to write parallel programs!

The *Skel* Library for Erlang

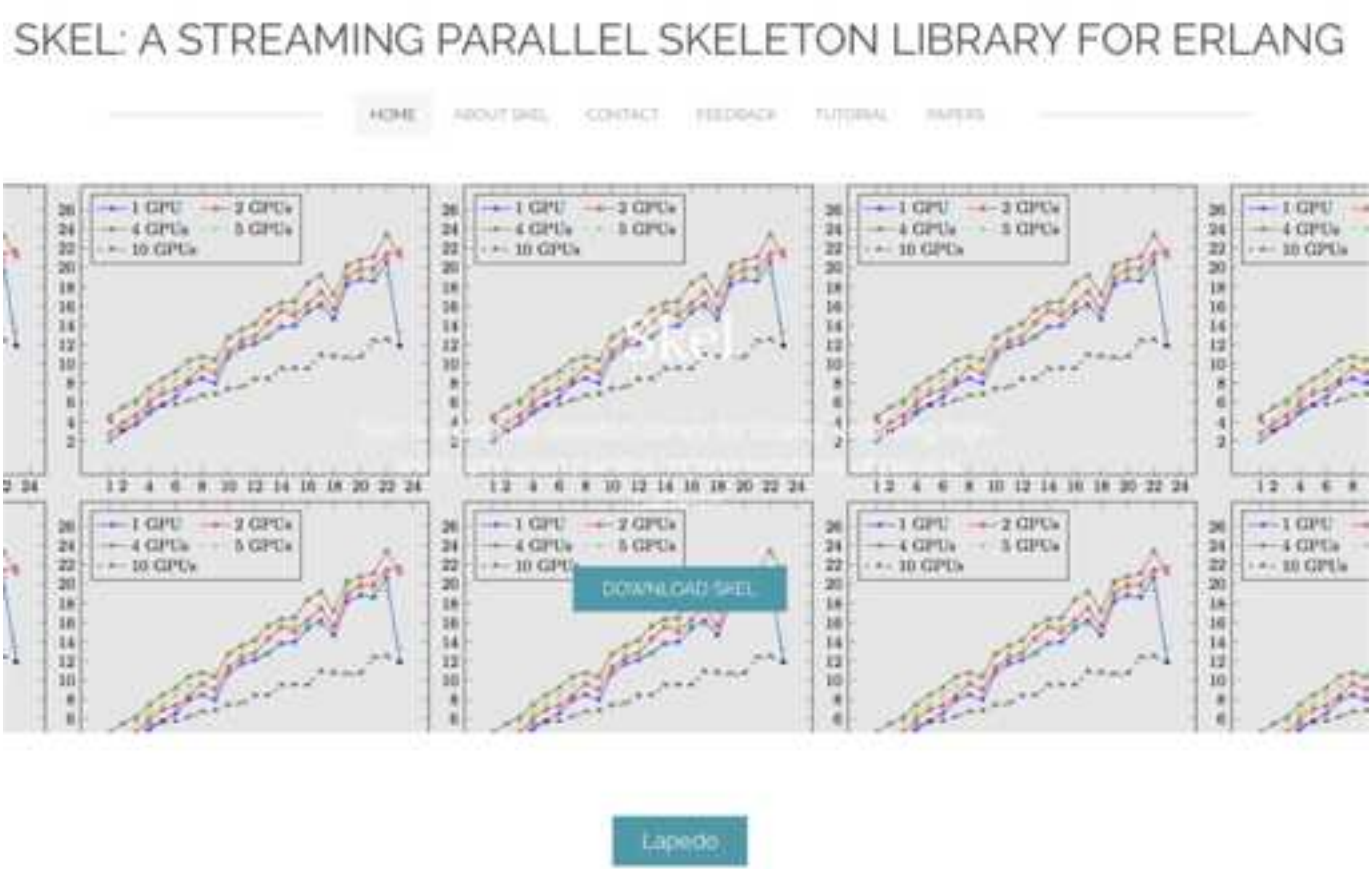
- Skeletons implement specific parallel patterns
  - Pluggable templates
- **Skel** is a new (AND ONLY!) Skeleton library in Erlang
  - map, farm, reduce, pipeline, feedback
  - instantiated using **skel:do**
- ***Fully Nestable***
- **A DSL for parallelism**

Skel.weebly.com

https://github.com/ParaPhrase/skel

```
OutputItems = skel:do(Skeleton, InputItems).
```

www.skel.weebly.com



Skel Overview

- Structured parallel programming framework
  - “palette” of skeletons/parallel patterns available (almost) arbitrary composition supported
  - very hard to implement arbitrary parallel schema
- Streaming parallel programming framework
  - primitive support to process streams of tasks

Skel. Who? Where?

- Main developers at St Andrews
  - Me ☺, Adam Barwell, Sam Elliott, Vladimir Janjic, Kevin Hammond, …
- Used on EU projects, ParaPhrase (FP7; ended 2015)
- Contributions from Open-source community
  - Poland, Sheffield, Hungary, …
- Used for all sorts of problems:
  - EMAS, image processing, GPU programming,
- Code available on git: https://github.com/ParaPhrase/skel

ParaPhrase / skel

Watch 20 | Star 191 | Fork 26

Code | Issues 1 | Pull requests 5 | Projects 0 | Security | Insights

Join GitHub today

GitHub is home to over 36 million developers working together to host and review code, manage projects, and build software together.

Sign up

Dismiss

A Streaming Process-based Skeleton Library for Erlang

10 commits | 2 branches | 0 releases | 3 contributors | BSD-3-Clause

Branch: master | New pull request | Find File | Clone or download

adbarwell Added top-level farm and pipe interface functions. Latest commit b755de9 on 25 Nov 2015

doc	API Documentation and Tutorial added; Map and Decomp changed to Map a…	6 years ago
examples	API Documentation and Tutorial added; Map and Decomp changed to Map a…	6 years ago
include	Added hyb_map and hyb_farm skeletons	5 years ago
priv	Missing default args	7 years ago
src	Added top-level farm and pipe interface functions.	4 years ago
tutorial	API Documentation and Tutorial added; Map and Decomp changed to Map a…	6 years ago
.gitignore	Initial Release	7 years ago
LICENCE	Initial Release	7 years ago
Makefile	API Documentation and Tutorial added; Map and Decomp changed to Map a…	6 years ago
README.md	API Documentation and Tutorial added; Map and Decomp changed to Map a…	6 years ago
rebar	Examples now compile. TODO: find lmb.hrl	7 years ago
rebar.config	API Documentation and Tutorial added; Map and Decomp changed to Map a…	6 years ago

README.md

skel

A Streaming Process-based Skeleton Library for Erlang

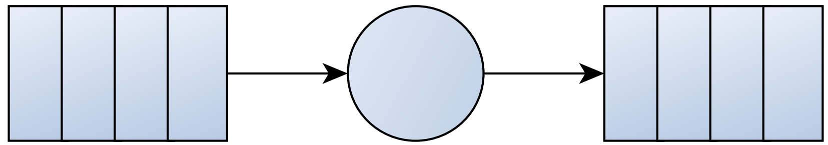
Usage

`make` to compile the library source

Skeletons supported

- Stream parallel
  - Pipeline (computations in stages)
  - Farm (Embarassingly parallel…)
  - Feedback loop (route back output tasks satisfying condition)
- Data parallel
  - Map (applying function over all items in stream)
  - Reduce ("summing" up items in a stream)
- Non primitive
  - Divide and conquer
  - Stencil
  - …

The Concept of Parallelism



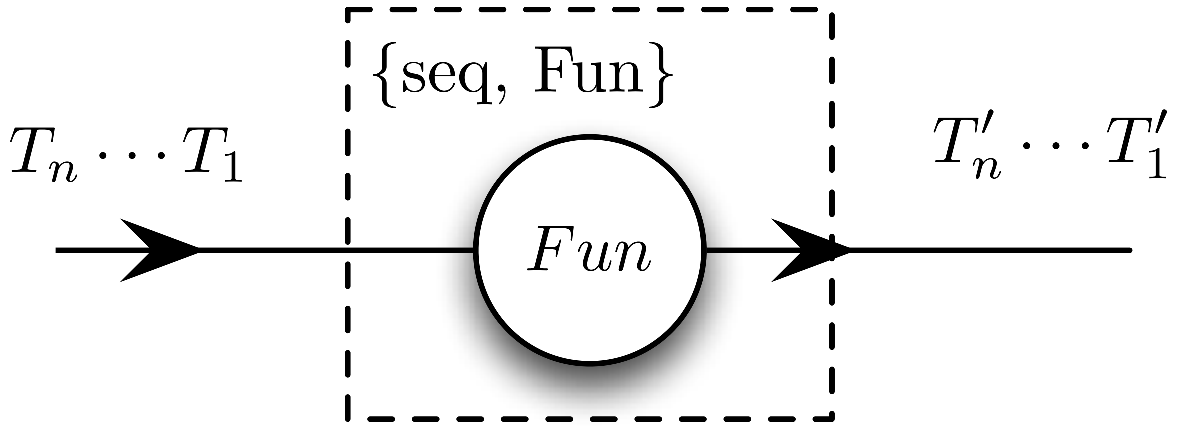
- Input queue
  - Feeds tasks from an input stream
  - Connected to the ‘parallel activity’
- Parallel activity body
  - Processes each item appearing onto the input stream
  - Delivers a result to the output stream
  - Maybe sequential *or* parallel
- Output queue
  - Receives output results from parallel body
  - Possibly connected to the input of another parallel activity
  - May not preserve order

Loading skel

```
$ cd skel
$ make examples
$ make console
```

Wrapping things up: the Seq skeleton

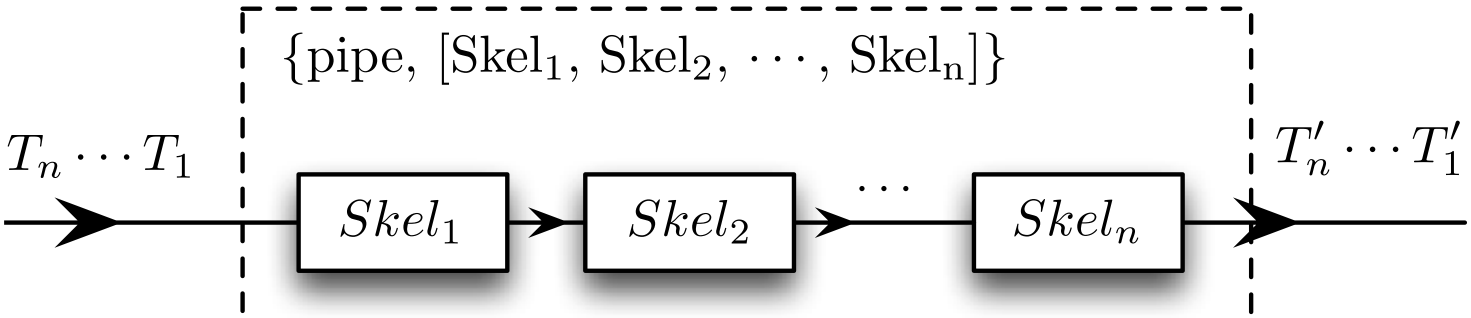
- Used to wrap up a function




```
skel:do([{seq, fun(X) -> X+1 end}],
                  [1,2,3,4,5,6,7,8,9,10]).
% -> [2,3,4,5,6,7,8,9,10,11]
```

Parallel Pipeline Skeleton

- Each stage of the pipeline can be executed in parallel
- The input and output are streams




```
skel:do([{pipe,[Skel1, Skel2,…,SkelN]}], Inputs).
```

Pipeline – 1st example

[1,2,3,4,5] → fun(X) -> X+1 → [2,3,4,5,6]

```
> cd skel
> make console
erl> skel:do(
   [{pipe,
     [{seq, fun(X) -> X+1 end}]}], [1,2,3,4,5]).
```

Pipeline – 2nd example

[1,2,3,4,5] → fun(X) -> X+1 → fun(X) -> X+1 → [3,4,5,6,7]

```
> cd skel
> make console
erl> skel:do(
    [{pipe,
        [{seq, fun(X) -> X+1 end},
         {seq, fun(Y) -> Y+1 end}]}],
                                  [1,2,3,4,5]).
```

Constructing Pipeline Workers

```
> Stage1 = {seq, fun(X) ->
                {fib:fib(X), X} end}.

> Stage2 = {seq, fun({X,Y}) ->
                     fib:fib(Y) end}.
```

Using a pipeline

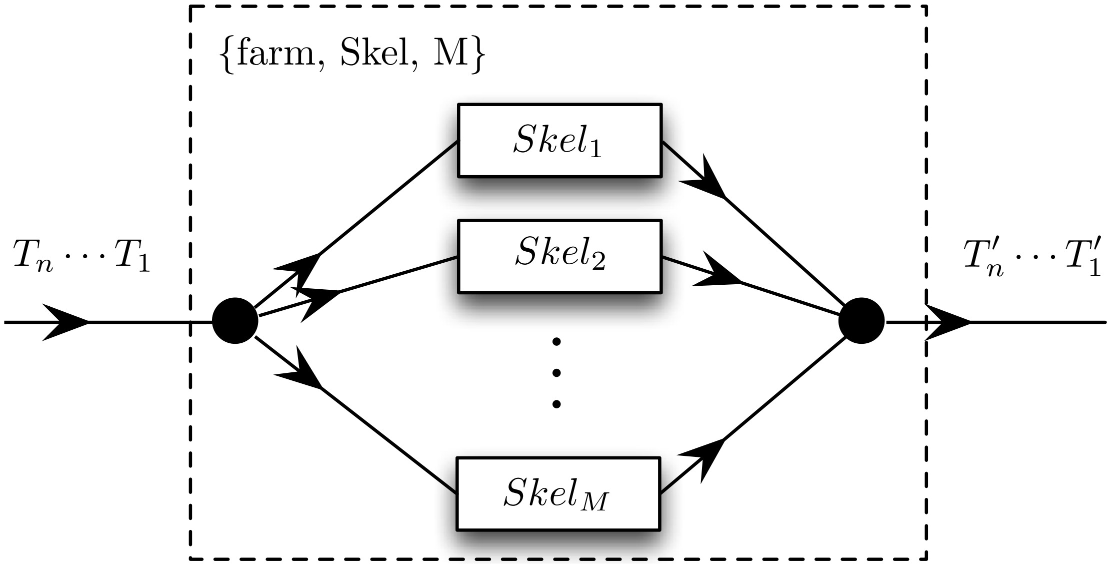
```
> InputsP = lists:duplicate(8, 27).

> {T2P, V2P} = timer:tc(fun() ->
      skel:do([{pipe,
                [Stage1,Stage2]}],
                        InputsP) end).
```

Creating a sequential pipeline

```
> Stage1S = fun(X) -> {fib:fib(X), X}
end.

> Stage2S = fun({X,Y}) -> fib:fib(Y)
end.

> {T1P, V1P} = timer:tc(fun() ->
  [ Stage2S(Stage1S(X)) || X <- InputsP
] end).

> T1P / T2P.
```

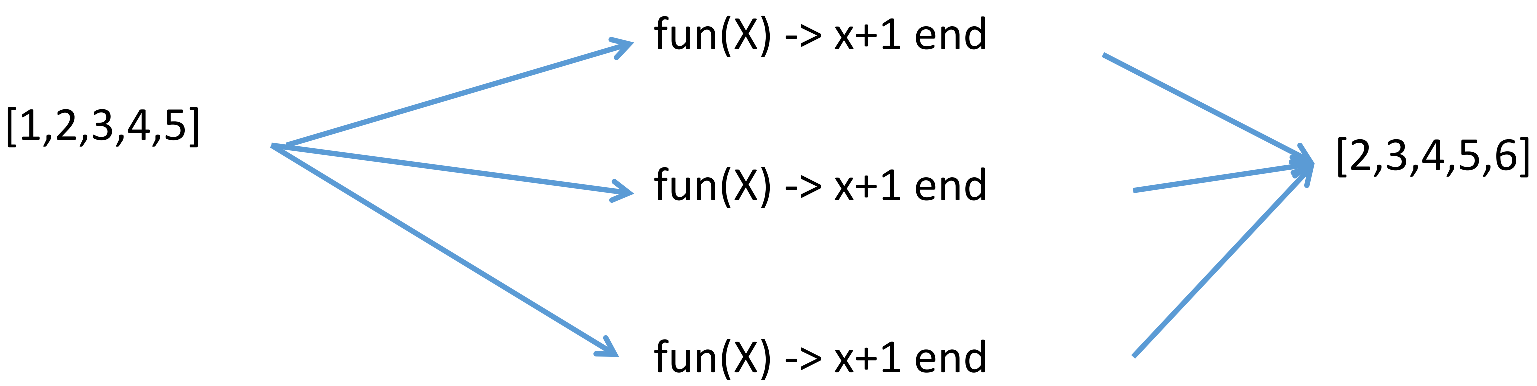
Farm Skeleton

- Each worker is executed in parallel
- A bit like a 1-stage pipeline




```
skel:do([{farm, Skel, M}], Inputs).
```

Farm – 1st Example




```
> cd skel
> make console
erl> skel:do([{farm, [fun(X) -> X+1 end], 2}], [1,2,3,4,5]).
```

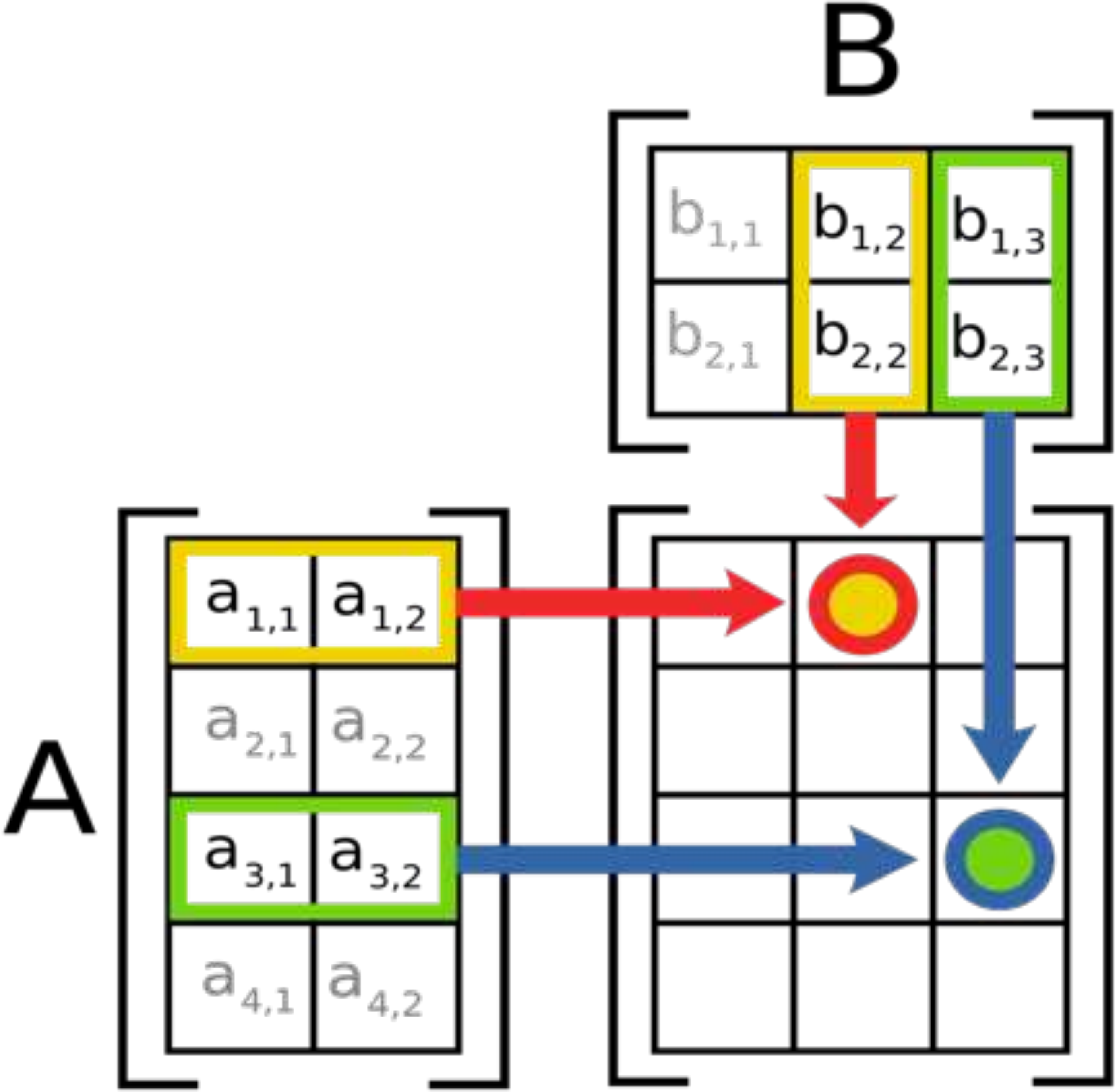
Using a Farm

```
> Payload = {seq, fun(X) ->
fib:fib(X) end}.

> Inputs = lists:duplicate(8, 27).

> NumberWorkers = 4.

> skel:do([{farm, [Payload],
NumberWorkers}], Inputs).
```

Timing the farm

```
> {T1, V1} = timer:tc(fun() ->
                    skel:do([{farm, [Payload],
                    1}], Inputs).

> {T2, V2} =  timer:tc(fun() ->
                    skel:do([{farm, [Payload],
                    NumberWorkers}], Inputs).

> T1 / T2.
```

Matrix Multiplication



https://en.wikipedia.org/wiki/Matrix_multiplication#/media/File:Matrix_multiplication_diagram_2.svg

Sequential Implementation

```
main(Nrows, S) ->
MatrixA = randmat(Nrows, Nrows, S),
MatrixB = randmat(Nrows, Nrows, S),
productMat(MatrixA, MatrixB).

productMat(MatrixA, MatrixB) ->
       mult(rows(MatrixA), cols(MatrixB)).

mult([],_) -> [];
mult([R|Rows], Cols) ->
       [lists:map(fun(X) ->
                      multSum(R, X) end, Cols)
                     | mult(Rows, Cols)].

multSum(R, C) ->
       lists:sum([A*B || {A,B} <- lists:zip(R,C) ] ) .
```

Naïve parallelisation

```
mult_par_2([],_) -> [];
mult_par_2([R|Rows], Cols) ->
   [skel:do([{farm, [{seq, fun(C) ->

    multSum(R,C) end}],10}], Cols) |
                                  mult(Rows, Cols)].
```

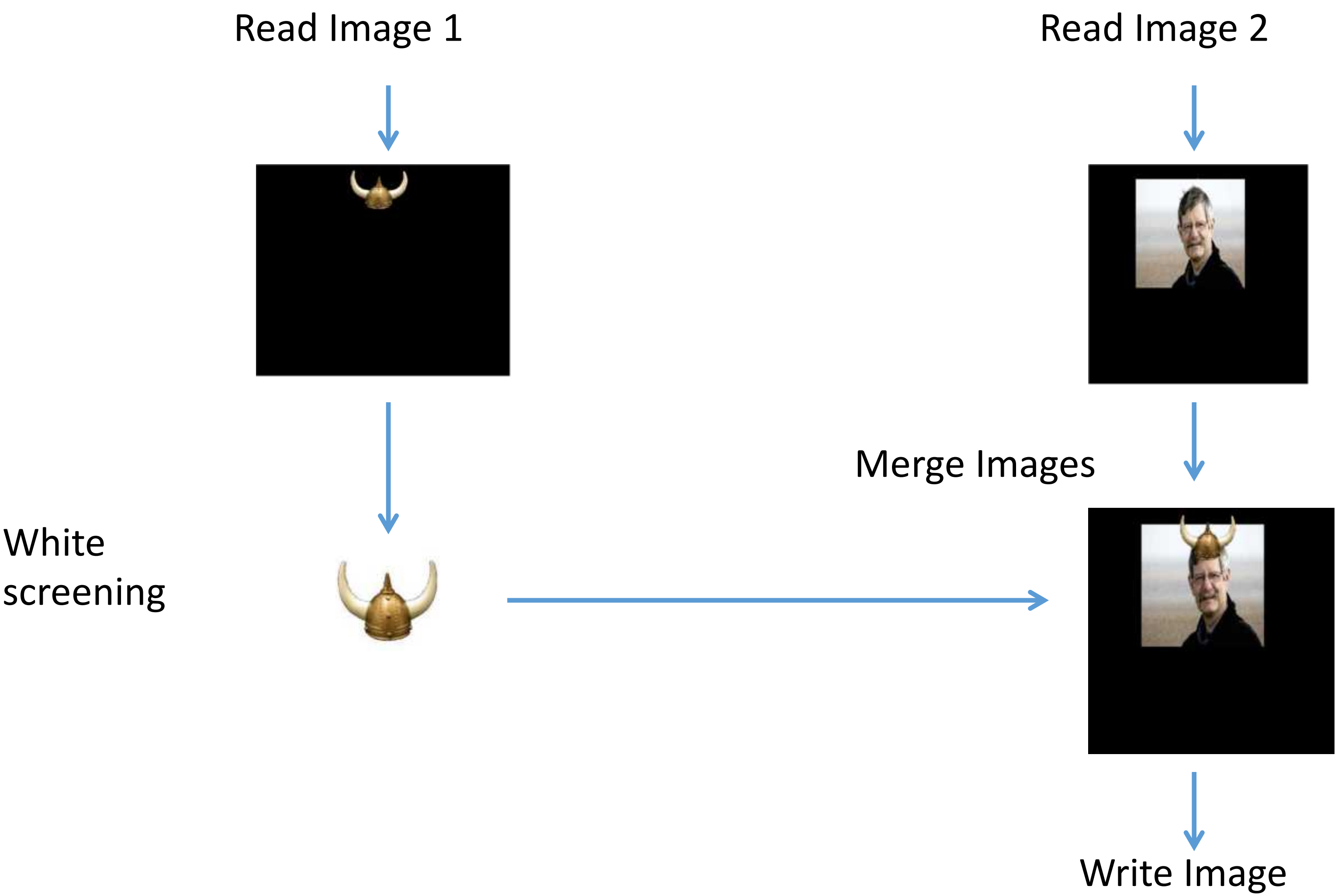
Better parallelisation

```
mult_par_1(Rows, Cols) ->
  skel:do([{farm, [{seq, fun(R) ->
          lists:map(fun(C) ->
             multSum(R,C) end, Cols) end}], 10}],
                                  Rows).
```

```
multSum_par_1(R,C) ->
  lists:sum(skel:do([{farm, [{seq, fun({A,B}) -> A*B
end}],10}], lists:zip(R,C))).
```

```
multSum_par_2(R,C) ->
  skel:do([{reduce, fun(A, B) -> A*B end, fun id/1}],
lists:zip(R,C)).
```

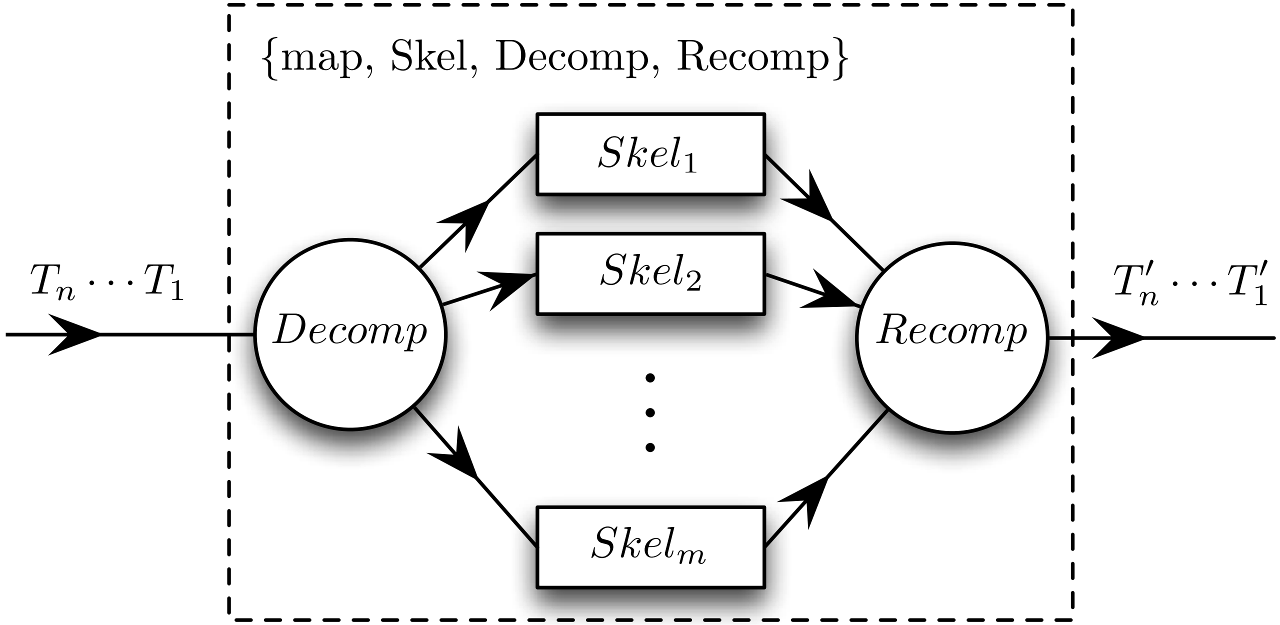
Image Processing Example



Basic Erlang Structure

```
[ writeImage(convertMerge(readImage(X)))
                                 || X <- Images() ]

readImage({In1, in2, out) ->
      …
      { Image1, Image2, out}.

convertImage({Image1, Image2, out}) ->
      Image1P = whiteScreen(Image1),
      Image2P = mergeImages(Image1, Image2),
      {Image2P, out}.

writeImage({Image, Out}) -> …
```

Map Skeleton




```
skel:do([{map, Skel, Decomp,
Recomp}], Inputs).
```

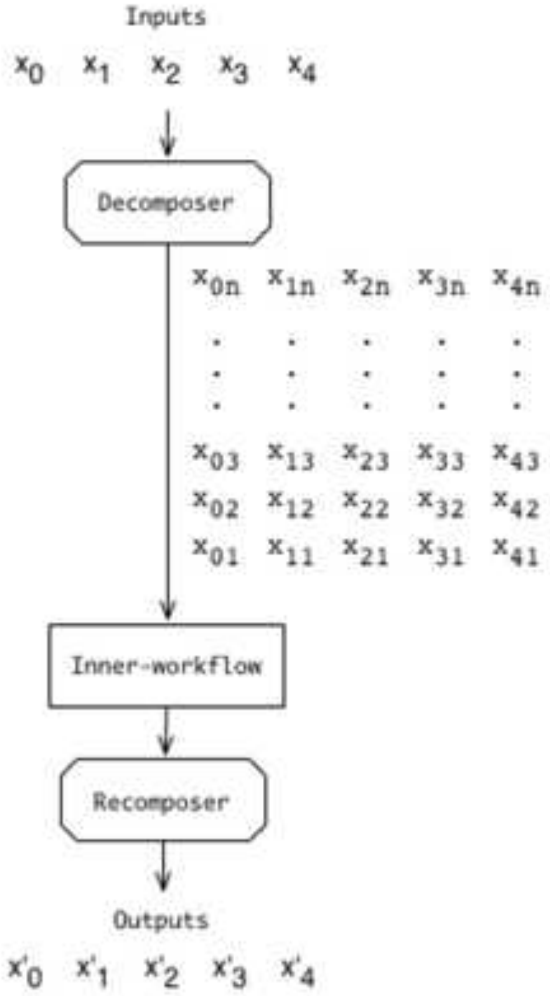
Introduce Map

```
{seq, Expr}
```

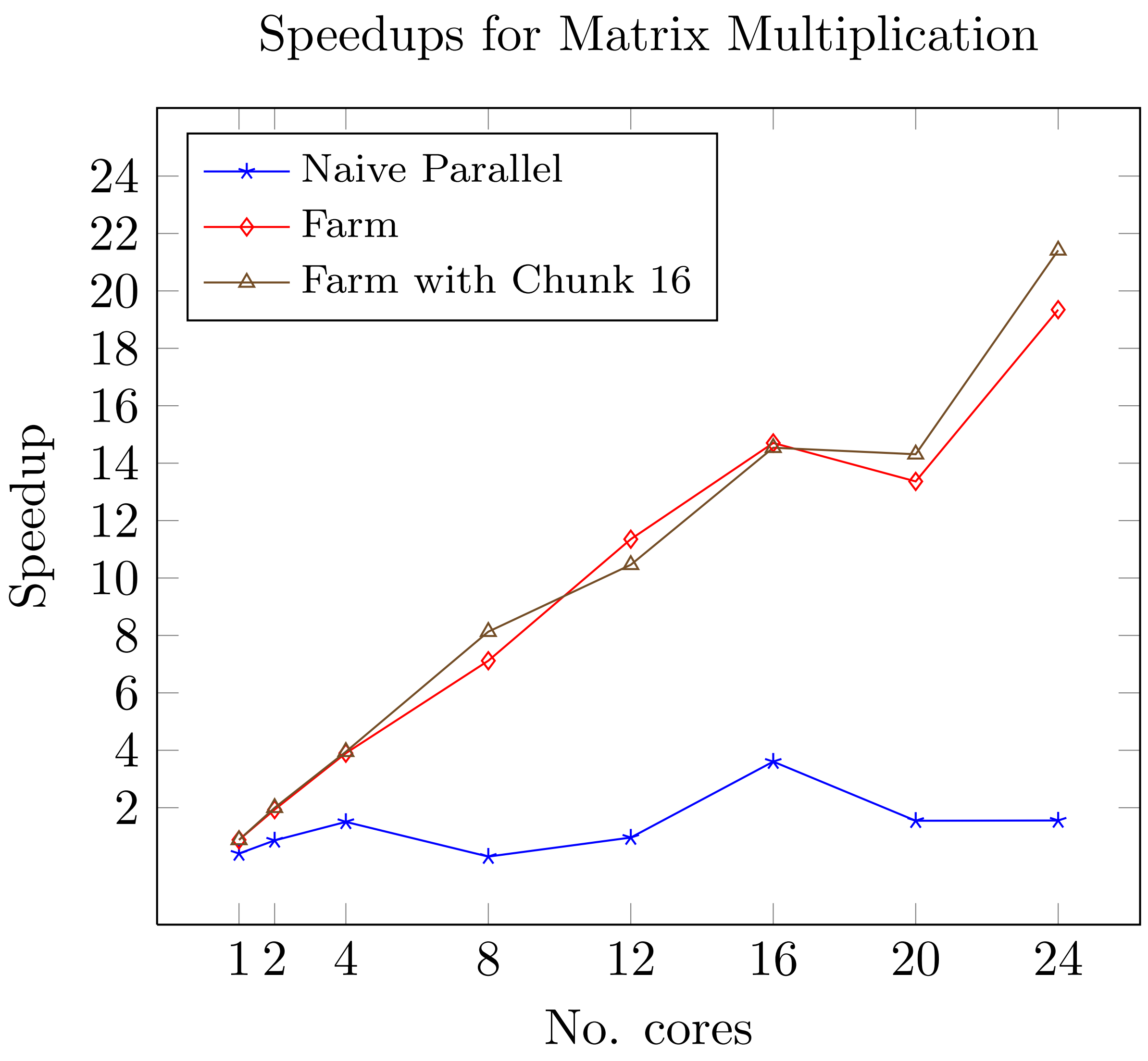
```
{map, {seq, Expr'}, fun ?MODULE:split/1,
                    fun ?MODULE:recomp/1}
```

Expr', `split` and `recomp` are arguments to the refactoring

Cluster Skeleton



Using The Right Pattern Matters



Cost Models

$$T_{C_{pipeline}}(L) = max_{i=1..m}(T_{stage_i}(L)) + T_{copy}(L)$$

$$\begin{aligned} T_{C_{map}}(L) &= T_{distrib}(N_w, L) + \frac{T_{Fun}(L)}{Min(N_p, N_w)} + T_{gather}(N_w, L) \\ & \quad \texttt{where} \;\; N_W = npartitions(L) \end{aligned}$$

$$T_{C_{farm}}(N_w, L) = max\{T_{emitter}(N_{p,N_w}, L), \frac{T_{Fun}(L)}{Min(N_p, N_w)}, T_{collector}(N_w, L)\}$$

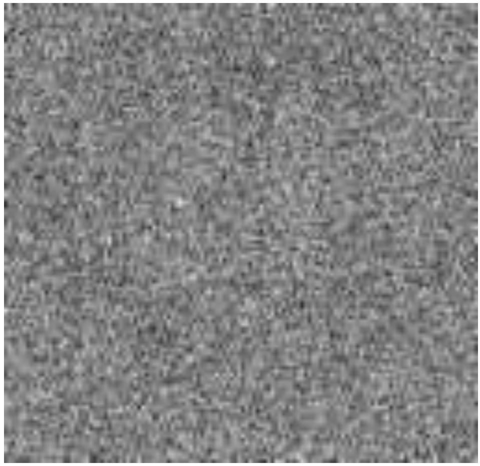
(2)

Case Study: De-Noising

Original
Baboon standard
test image
1024x1024

10% impulsive noise

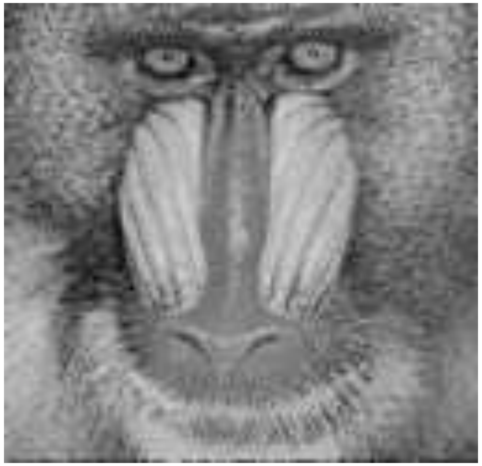
50% impulsive noise



90% impulsive noise

Restored

PNSR 43.29dB MAE 0.35



PNSR 32.75dB MAE 2.67

PNSR 23.4 MAE 11.21

Stage 1: Introduce Pipeline

```
denoise(Ims) -> [  filter (geoRef ( Im ) ) || Im <- Ims ].
```

```
denoise(Ims) -> skel:run([{pipe, [{seq, fun ?MODULE:geoRef/1},
                                  {seq, fun ?MODULE:f filter/1}]}], Ims).
```

$$(171 + 466) * 1024$$

652288

$$(Max(171, 466) * 1024)$$

477184

Stage 1: Introduce Par Map

```
denoise(Ims) -> skel:run([{pipe, [{seq, fun ?MODULE:geoRef/1},
                         {seq, fun ?MODULE:f filter/1}]}], Ims).
```

```
denoise(Ims) ->  skel:run([{pipe, [{seq, fun ?MODULE:geoRef/1},
                  {parmap, [{seq,fun ?MODULE:filter'/1}],
                           fun ?MODULE:partition/1,
                           fun ?MODULE:combine/1}]}], Ims).
```

$$(Max(171, 466) * 1024)$$ 477184

*(Max(171, (0.001 + 29 + 0.001)) * 1024)* 175104

Stage 1: Introduce Task Farm

```
denoise(Ims) ->  skel:run([{pipe, [{seq, fun ?MODULE:geoRef/1},
                      {parmap, [{seq,fun ?MODULE:filter'/1}],
                               fun ?MODULE:partition/1,
                               fun ?MODULE:combine/1}]}], Ims).
```

```
denoise(Ims) -> skel:run([{pipe,[{farm,[{seq,fun ?MODULE:geoRef}],
                   Nw}, {parmap,  [{seq, fun ?MODULE:filter'}],
                            fun ?MODULE:partition/1,
                            fun ?MODULE:combine/1}]}], Ims).
```

*(Max(171, (0.001 + 29+ 0.001)) * 1024)*

175104

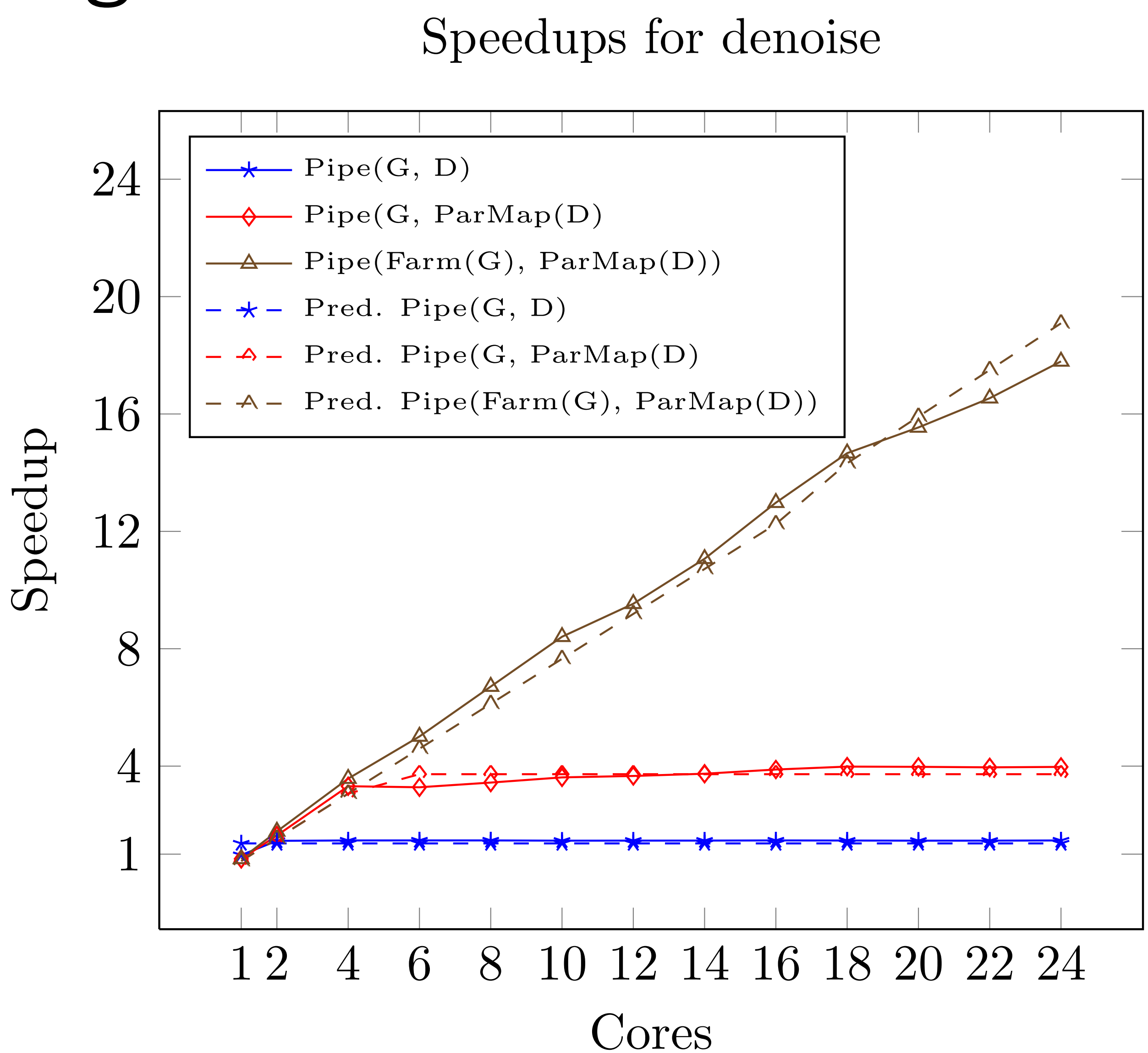
*(Max(171/8, (0.001 + 29+ 0.001)) * 1024)*

29698

Performance Results

- 8 Core Dell PowerEdge
  - 24 Core Dual AMD Opteron 6176 2.3GHz
  - 12GB RAM
  - 6MB L2 Cache
  - Linux 2.6.18
  - Erlang 5.9.1 R15
- Runtimes averaged over 10 runs
- Input of 1024 images

Predicted vs. Actual Speedups for Erlang



Thank you!

cmb21@st-andrews.ac.uk

@chrismarkbrown