Erlang as a framework for parallel/distributed programming.

Marco Stronati
marco.stronati@gmail.com

2010

- personal inclination towards functional programming (strong typed!)

- personal inclination towards functional programming (strong typed!)
- much work on haskell concurrency/parallelism (way too much material) but no distribution

- personal inclination towards functional programming (strong typed!)
- much work on haskell concurrency/parallelism (way too much material) but no distribution
- Erlang...

- personal inclination towards functional programming (strong typed!)
- much work on haskell concurrency/parallelism (way too much material) but no distribution
- Erlang... I think I found something...

- personal inclination towards functional programming (strong typed!)
- much work on haskell concurrency/parallelism (way too much material) but no distribution
- Erlang... I think I found something...
- *Talk is cheap. Show me the code*

- personal inclination towards functional programming (strong typed!)
- much work on haskell concurrency/parallelism (way too much material) but no distribution
- Erlang... I think I found something...
- *Talk is cheap. Show me the code* put together a proof of concept

- personal inclination towards functional programming (strong typed!)
- much work on haskell concurrency/parallelism (way too much material) but no distribution
- Erlang... I think I found something...
- *Talk is cheap. Show me the code* put together a proof of concept
- it'll be nice to make a comparative benchmark (both on time and loc)

Erlang - Hystory

82-85 Experiments. The language must be high level, symbolic (Lisp , Prolog ...).

85-86 The language must contain primitives for concurrency and error recovery.

87-89 Erlang was developed.

89-97 Erlang grows both in users base and code base terms.

98 Erlang open sourced.

today Erlang, together with libraries and the real-time distributed database Mnesia, forms the Open Telecom Platform (OTP).

Erlang - Higher Order

Treat your functions like your integers!

```
f(10,fun(x) -> x+1 end).
```

Erlang - Higher Order

Treat your functions like your integers!

```
f(10,fun(x) -> x+1 end).
```

Functions are first-order values $\Rightarrow$ can be passed/returned to/by functions.

Erlang - Higher Order

Treat your functions like your integers!

```
f(10,fun(x) -> x+1 end).
```

Functions are first-order values $\Rightarrow$ can be passed/returned to/by functions.
This means functions can be:

1. modified by other functions
2. easily passed over the network

Erlang - Higher Order

Treat your functions like your integers!

```
f(10,fun(x) -> x+1 end).
```

Functions are first-order values $\Rightarrow$ can be passed/returned to/by functions.
This means functions can be:

1. modified by other functions
2. easily passed over the network

```
spawn(test@fujim1,
                  f,
      [10, fun(x) -> x+1 end]).
```

Erlang - Higher Order

Treat your functions like your integers!

```
f(10,fun(x) -> x+1 end).
```

Functions are first-order values ⇒ can be passed/returned to/by functions.
This means functions can be:

1. modified by other functions
2. easily passed over the network

```
spawn(test@fujim1,
                  f,
      [10, fun(x) -> x+1 end]).
```

Code is not first-class ⇒ modules should be.

Erlang - Types and Pattern Matching

Erlang type system is dynamic $\Rightarrow$ compile right, run badly.

Erlang - Types and Pattern Matching

Erlang type system is dynamic $\Rightarrow$ compile right, run badly.
Few types:
integer, float, bool, atom (`eof`), fun (fun...end), pid (`<0.39.0>`)

Erlang - Types and Pattern Matching

Erlang type system is dynamic $\Rightarrow$ compile right, run badly.
Few types:
integer, float, bool, atom (`eof`), fun (fun...end), pid (`<0.39.0>`)
lists (`[1,2.0,eof]`), tuple (`{1,2.0,eof}`)

Erlang - Types and Pattern Matching

Erlang type system is dynamic ⇒ compile right, run badly.
Few types:
integer, float, bool, atom (`eof`), fun (`fun...end`), pid (`<0.39.0>`)
lists (`[1,2.0,eof]`), tuple (`{1,2.0,eof}`)

```
1  server_protocol(Msg)->
2    case Msg of
3      start -> start_service(),
4               send({start,ack});
5      stop -> stop_service(),
6              send({stop,ack});
7      {idx,data} -> send({idx,service(data)})
8    end.
```

Erlang - Types and Pattern Matching

Erlang type system is dynamic ⇒ compile right, run badly.
Few types:
integer, float, bool, atom (`eof`), fun (`fun...end`), pid (`<0.39.0>`)
lists (`[1,2.0,eof]`), tuple (`{1,2.0,eof}`)

```
1  server_protocol(Msg)->
2    case Msg of
3      start -> start_service(),
4               send({start,ack});
5      stop -> stop_service(),
6              send({stop,ack});
7      {idx,data} -> send({idx,service(data)})
8    end.
```

Dynamic Typing:

Erlang - Types and Pattern Matching

Erlang type system is dynamic ⇒ compile right, run badly.
Few types:
integer, float, bool, atom (`eof`), fun (`fun...end`), pid (`<0.39.0>`)
lists (`[1,2.0,eof]`), tuple (`{1,2.0,eof}`)

```
1  server_protocol(Msg)->
2    case Msg of
3      start -> start_service(),
4               send({start,ack});
5      stop -> stop_service(),
6              send({stop,ack});
7      {idx,data} -> send({idx,service(data)})
8    end.
```

Dynamic Typing:

⇒ expoit it: rapid prototyping.
`[1,2.0,334,17.2]`

Erlang - Types and Pattern Matching

Erlang type system is dynamic ⇒ compile right, run badly.
Few types:
integer, float, bool, atom (`eof`), fun (`fun...end`), pid (`<0.39.0>`)
lists (`[1,2.0,eof]`), tuple (`{1,2.0,eof}`)

```
1  server_protocol(Msg)->
2    case Msg of
3      start -> start_service(),
4               send({start,ack});
5      stop -> stop_service(),
6              send({stop,ack});
7      {idx,data} -> send({idx,service(data)})
8    end.
```

Dynamic Typing:

⇒ expoit it: rapid prototyping.
`[1,2.0,334,17.2]`

⇒ avoid it: production use.
define (`-def` and `-type`) and check (`dialyzer`) your types for serious projects.
`-type number :: float() | integer()`

Erlang - Concurrent/Distributed

Implements asynchronous message-passing model through light-weight processes.

Erlang - Concurrent/Distributed

Implements asynchronous message-passing model through light-weight processes.
Share-nothing paradigm $\Rightarrow$ lock-free $\Rightarrow$ very scalable.

Erlang - Concurrent/Distributed

Implements asynchronous message-passing model through light-weight processes.
Share-nothing paradigm $\Rightarrow$ lock-free $\Rightarrow$ very scalable.

Message passing works seemlessly locally

Erlang - Concurrent/Distributed

Implements asynchronous message-passing model through light-weight processes.
Share-nothing paradigm $\Rightarrow$ lock-free $\Rightarrow$ very scalable.

Message passing works seemlessly locally

```
spawn(Module, Function, ArgumentList) -> pid()
```

Erlang - Concurrent/Distributed

Implements asynchronous message-passing model through light-weight processes.
Share-nothing paradigm $\Rightarrow$ lock-free $\Rightarrow$ very scalable.

Message passing works seemlessly locally

```
spawn(Module, Function, ArgumentList) -> pid()
```

or remotely

```
spawn(Node, Module, Function, ArgumentList) -> pid()
```

Erlang - Concurrent/Distributed

Implements asynchronous message-passing model through light-weight processes.
Share-nothing paradigm $\Rightarrow$ lock-free $\Rightarrow$ very scalable.

Message passing works seemlessly locally

```
spawn(Module, Function, ArgumentList) -> pid()
```

or remotely

```
spawn(Node, Module, Function, ArgumentList) -> pid()
```

Concurency follows the Actor model.

Erlang - Concurrent/Distributed example

```
1  rtt_server() ->
2  ...
3    Pids = map(fun(Node) ->
4                   spawn(Node,pmap,rtt_client,[self()])
5                end, Nodes),
6    Rtts = map(fun(Pid) ->
7                  Start = statistics(wall_clock),
8                  Pid ! Data,
9                  receive
10                   Rec when (Rec == Data) ->
11                             {_,Rtt} = statistics(
                                     wall_clock);
12                   _ -> io:format("ERROR")
13                 end,
14                 Rtt
15               end, Pids),
16 ...
```

Erlang - Concurrent/Distributed example

```
1  rtt_server() ->
2  ...
3    Pids = map(fun(Node) ->
4                   spawn(Node,pmap,rtt_client,[self()])
5                end, Nodes),
6    Rtts = map(fun(Pid) ->
7                  Start = statistics(wall_clock),
8                  Pid ! Data,
9                  receive
10                   Rec when (Rec == Data) ->
11                             {_,Rtt} = statistics(
                                     wall_clock);
12                   _ -> io:format("ERROR")
13                 end,
14                 Rtt
15               end, Pids),
16 ...
```

```
1  rtt_client(Master) ->
2      receive
3          Data -> Master ! Data
4      end.
```

Erlang - Soft real-time, extras

Easy to set timers:

Erlang - Soft real-time, extras

Easy to set timers:

```
1  rtt_client(Master) ->
2      Timeout = 6000, %ms
3      receive
4          Data -> Master ! Data;
5          _ -> io:format('error')
6      after
7          Timeout -> Master ! eof
8      end.
```

Erlang - Soft real-time, extras

Easy to set timers:

```
1  rtt_client(Master) ->
2      Timeout = 6000, %ms
3      receive
4          Data -> Master ! Data;
5          _ -> io:format('error')
6      after
7          Timeout -> Master ! eof
8      end.
```

- Network friendly: socket, pools, world(), ssh/ssl , mnesia etc

Erlang - Soft real-time, extras

Easy to set timers:

```
1  rtt_client(Master) ->
2      Timeout = 6000, %ms
3      receive
4          Data -> Master ! Data;
5          _ -> io:format('error')
6      after
7          Timeout -> Master ! eof
8      end.
```

- Network friendly: socket, pools, world(), ssh/ssl , mnesia etc
- Exception handling

Erlang - Soft real-time, extras

Easy to set timers:

```
1  rtt_client(Master) ->
2      Timeout = 6000, %ms
3      receive
4          Data -> Master ! Data;
5          _ -> io:format('error')
6      after
7          Timeout -> Master ! eof
8      end.
```

- Network friendly: socket, pools, world(), ssh/ssl , mnesia etc
- Exception handling
- Interoperativity with Java and C: no-call just messages in erlang-rts or binary

Erlang - Soft real-time, extras

Easy to set timers:

```
1  rtt_client(Master) ->
2      Timeout = 6000, %ms
3      receive
4          Data -> Master ! Data;
5          _ -> io:format('error')
6      after
7          Timeout -> Master ! eof
8      end.
```

- Network friendly: socket, pools, world(), ssh/ssl , mnesia etc
- Exception handling
- Interoperativity with Java and C: no-call just messages in erlang-rts or binary
- Native code compiler HYPE

Erlang - Soft real-time, extras

Easy to set timers:

```
1  rtt_client(Master) ->
2      Timeout = 6000, %ms
3      receive
4          Data -> Master ! Data;
5          _ -> io:format('error')
6      after
7          Timeout -> Master ! eof
8      end.
```

- Network friendly: socket, pools, world(), ssh/ssl , mnesia etc
- Exception handling
- Interoperativity with Java and C: no-call just messages in erlang-rts or binary
- Native code compiler HYPE
- Benchmark and Tracing infrastructure: Inviso

Erlang - Soft real-time, extras

Easy to set timers:

```
1  rtt_client(Master) ->
2      Timeout = 6000, %ms
3      receive
4          Data -> Master ! Data;
5          _ -> io:format('error')
6      after
7          Timeout -> Master ! eof
8      end.
```

- Network friendly: socket, pools, world(), ssh/ssl , mnesia etc
- Exception handling
- Interoperativity with Java and C: no-call just messages in erlang-rts or binary
- Native code compiler HYPE
- Benchmark and Tracing infrastructure: Inviso
- Low level operators (erlang embedded)

Erlang - Soft real-time, extras

Easy to set timers:

```
1  rtt_client(Master) ->
2      Timeout = 6000, %ms
3      receive
4          Data -> Master ! Data;
5          _ -> io:format('error')
6      after
7          Timeout -> Master ! eof
8      end.
```

- Network friendly: socket, pools, world(), ssh/ssl , mnesia etc
- Exception handling
- Interoperativity with Java and C: no-call just messages in erlang-rts or binary
- Native code compiler HYPE
- Benchmark and Tracing infrastructure: Inviso
- Low level operators (erlang embedded)
- Code hotload

Erlang - Hot code load

As a consequece of higher order, it is possible to hot load code:

```
1  loop(F) ->
2            receive
3              {request, Pid, Data} ->
4                        Pid ! F(Data),
5                        loop(F);
6              {change_code, F1} ->
7                  loop(F1)
8            end
```

Erlang - Hot code load

As a consequece of higher order, it is possible to hot load code:

```
1  loop(F) ->
2            receive
3              {request, Pid, Data} ->
4                        Pid ! F(Data),
5                        loop(F);
6              {change_code, F1} ->
7                  loop(F1)
8            end
```

```
1  Server ! {change_code, fun(I, J) ->
2                              do_something(...)
3                          end}
```

Parallel Map

Parallel Map: data parallel, single shot.

Parallel Map

Parallel Map: data parallel, single shot.

```
1 pmap(myfun,mydata)
```

Parallel Map

Parallel Map: data parallel, single shot.

```
1  pmap(myfun,mydata)
```

Erlang:

```
1  pmap(fun(X) -> X+1 end,[1,2,3]).
2  [2,3,4]
```

Parallel Map

Parallel Map: data parallel, single shot.

```
1  pmap(myfun,mydata)
```

Erlang:

```
1  pmap(fun(X) -> X+1 end,[1,2,3]).
2  [2,3,4]
```

```
1  pmap(fun(Vect) ->
2           map(fun(X) ->
3               foreach(fun(_) ->
4                           math:erf(X)
5                       end,
6               seq(1,NCPU)),
7               X
8               end, Vect)
9       end, Data]),
10 ...Vect...
```

Pmap

1. Master dispatch jobs in parallel.

Pmap

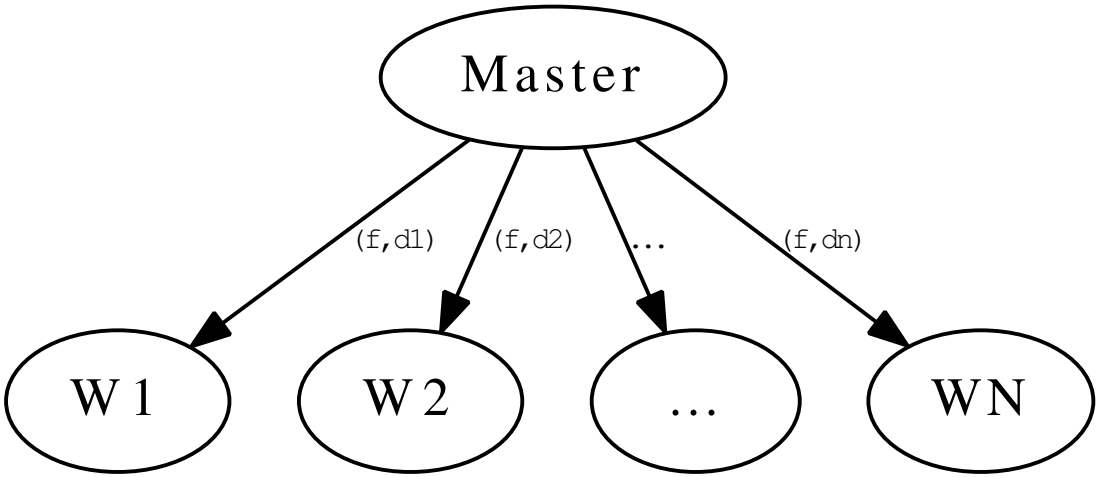
1. Master dispatch jobs in parallel.
2. Both function and data are sent together with an index.

Pmap

1. Master dispatch jobs in parallel.
2. Both function and data are sent together with an index.
3. The function is wrapped in a communication container.

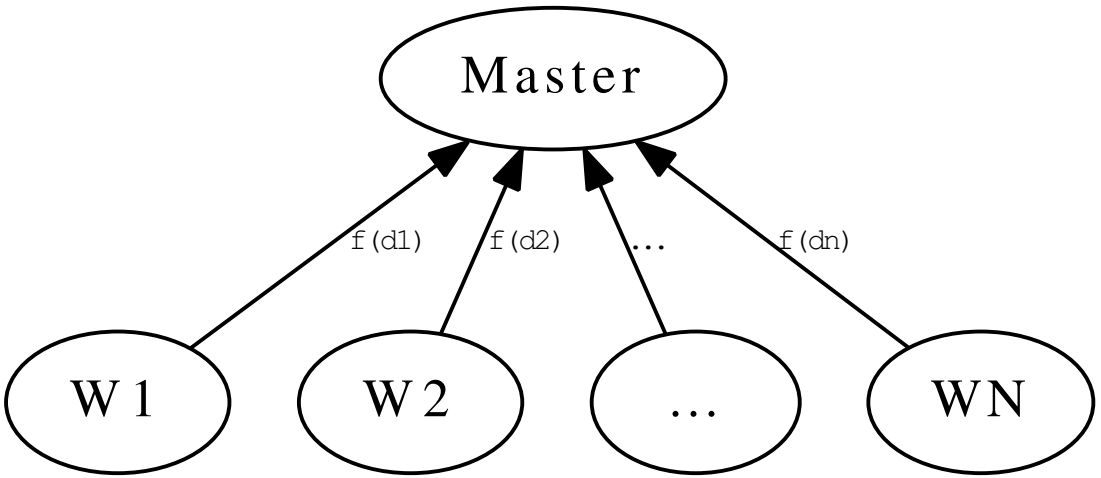
Pmap

1. Master dispatch jobs in parallel.
2. Both function and data are sent together with an index.
3. The function is wrapped in a communication container.



Pmap

Results are collected and sorted.



Pmap

1. Obtain n nodes.
2. Pack the data with an index.
3. Create couples {Node,Data}
4. Stages wait for data to process.

Pmap

1. Obtain n nodes.
2. Pack the data with an index.
3. Create couples {Node,Data}
4. Stages wait for data to process.

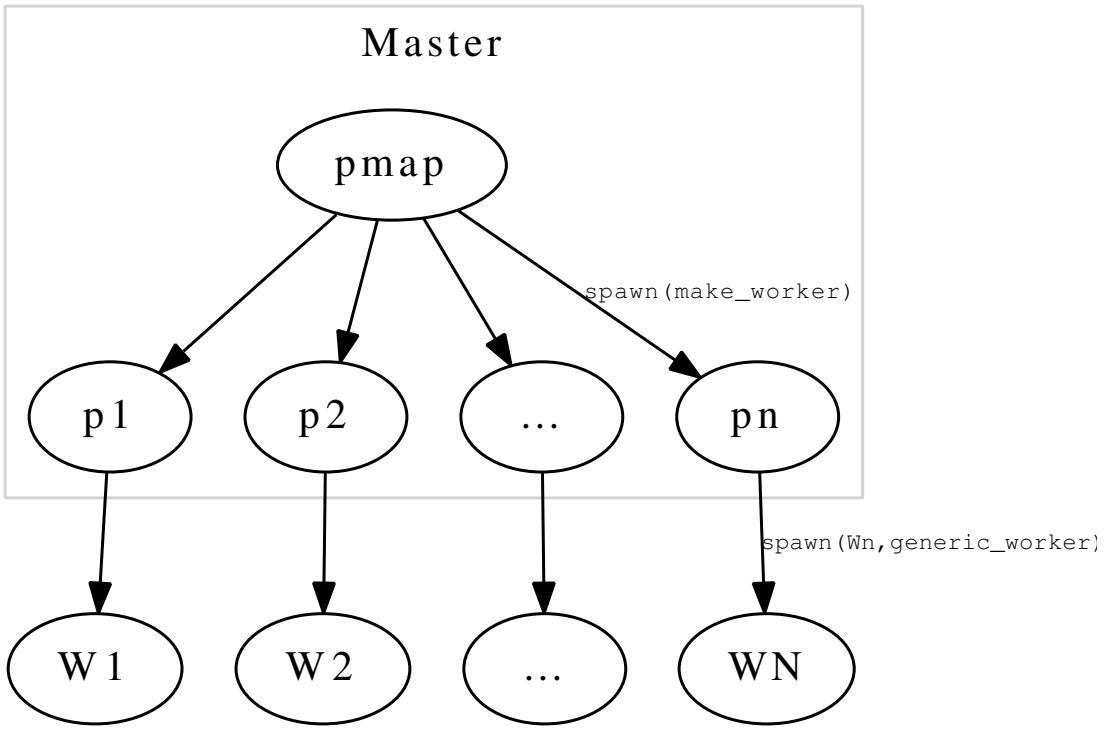
```
1  pmap(Function, Datas) ->
2      Nodes = get_nodes(length(Datas)),
3      Master = self(),
4      Indexed_data = zip(seq(1,length(Datas)),
5                                 Datas),
6      NDS = zip(Nodes,Indexed_data),
7  ...
```

Pmap

```
1  pmap(Function, Datas) ->
2  ...
3      lists:foreach(fun({Node,Data}) ->
4                            spawn(pmap,make_worker,
5                                  [Node, Function, Data,
                                       Master])
6                    end,NDS),
7      collect_results(length(Datas)).
```

Pmap

```
1  pmap(Function, Datas) ->
2  ...
3      lists:foreach(fun({Node,Data}) ->
4                            spawn(pmap,make_worker,
5                                  [Node, Function, Data,
                                       Master])
6                    end,NDS),
7      collect_results(length(Datas)).
```



Pmap

Each process (*thread*), spawns on a different node.

```
1  make_worker(Node, Function, Data, Master) ->
2      spawn(Node, pmap, generic_worker,
3            [Function,Data,Master]).
```

Pmap

Each process (*thread*), spawns on a different node.

```
1  make_worker(Node, Function, Data, Master) ->
2      spawn(Node, pmap, generic_worker,
3            [Function,Data,Master]).
```

Function deployed on workers with its very complex wrapper...

```
1  generic_worker(Fun,{Idx,Data},Master) ->
2      Master ! {Idx,Fun(Data)}.
```

Pmap

Each process (*thread*), spawns on a different node.

```
1  make_worker(Node, Function, Data, Master) ->
2      spawn(Node, pmap, generic_worker,
3            [Function,Data,Master]).
```

Function deployed on workers with its very complex wrapper...

```
1  generic_worker(Fun,{Idx,Data},Master) ->
2      Master ! {Idx,Fun(Data)}.
```

All pmap implementation $\sim$ 50 loc

Pmap

Each process (*thread*), spawns on a different node.

```
1  make_worker(Node, Function, Data, Master) ->
2      spawn(Node, pmap, generic_worker,
3            [Function,Data,Master]).
```

Function deployed on workers with its very complex wrapper...

```
1  generic_worker(Fun,{Idx,Data},Master) ->
2      Master ! {Idx,Fun(Data)}.
```

All pmap implementation $\sim$ 50 loc
Testing/tracing infrastructure $\sim$ 150 loc

Pipeline

Pipeline: one function per stage, stream of data.

Pipeline

Pipeline: one function per stage, stream of data.

```
1 create(fun1,fun2,...,fn).
2 run(data1,data2,...,datan).
```

Pipeline

Pipeline: one function per stage, stream of data.

```
1  create(fun1,fun2,...,fn).
2  run(data1,data2,...,datan).
```

Erlang:

```
1  Pids = create([fun(X) -> X+1 end, fun(X) -> X*X end]),
2  Res = run([1,2,3,4,5], hd(Pids)).
3  [4,9,16,25,36]
```

Pipeline

Pipeline: one function per stage, stream of data.

```
1  create(fun1,fun2,...,fn).
2  run(data1,data2,...,datan).
```

Erlang:

```
1  Pids = create([fun(X) -> X+1 end, fun(X) -> X*X end]),
2  Res = run([1,2,3,4,5], hd(Pids)).
3  [4,9,16,25,36]
```

```
1  Pids = create(duplicate(Stage_Number,
2                          (fun(X) ->
3                            Matrix = duplicate2(50,2.0),
4                            multiply_matrix(duplicate(NCPU,
                                  Matrix)),
5                            X
6                           end)),
7  Res = run(duplicate(NData,duplicate2(DData,2.0)),
8            hd(Pids))
```

Pipeline

1. Master deploy function and pid of the successor to each stage

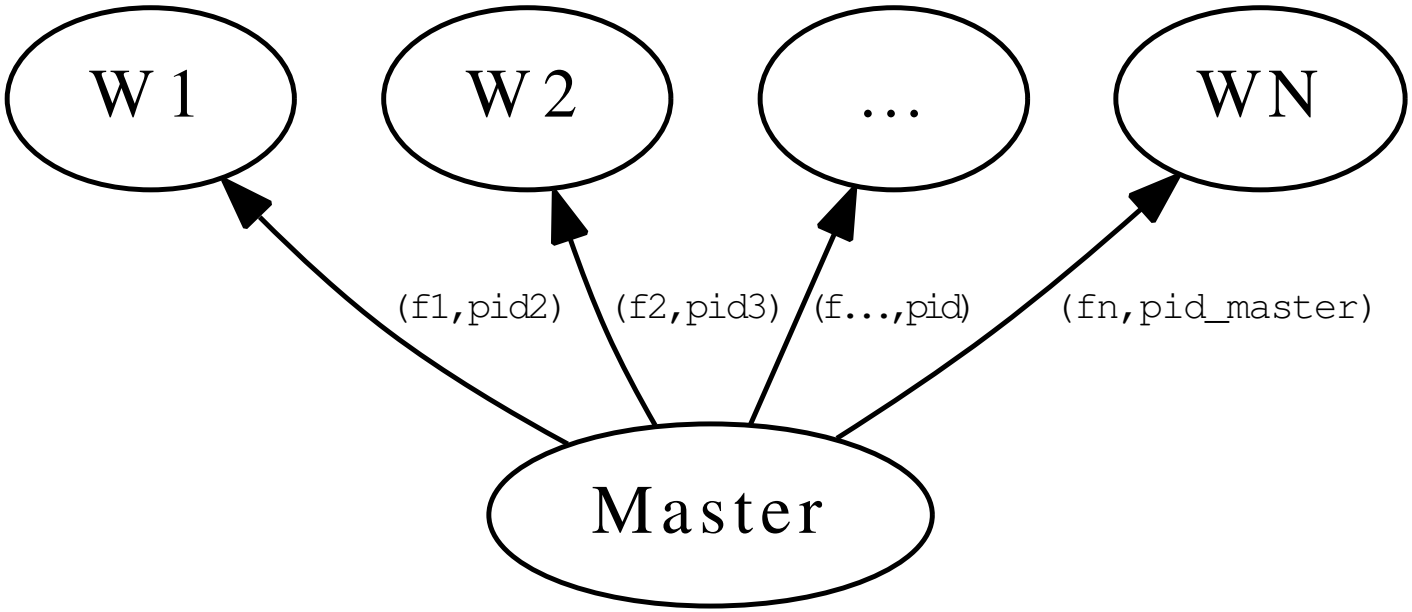
Pipeline

1. Master deploy function and pid of the successor to each stage
2. Stages wait for data to process.

Pipeline

1. Master deploy function and pid of the successor to each stage
2. Stages wait for data to process.

Ideally:



Pipeline

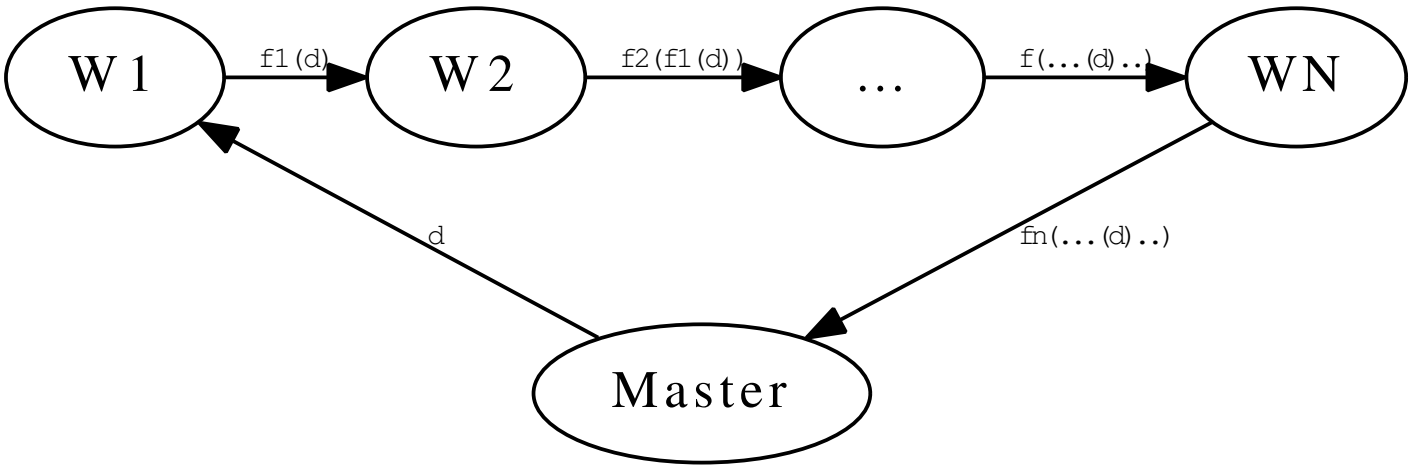
1. Master feeds data to the first stage.

Pipeline

1. Master feeds data to the first stage.
2. Each stage applies its function to the data and sends the result to its successor.

Pipeline

1. Master feeds data to the first stage.
2. Each stage applies its function to the data and sends the result to its successor.



Pipeline

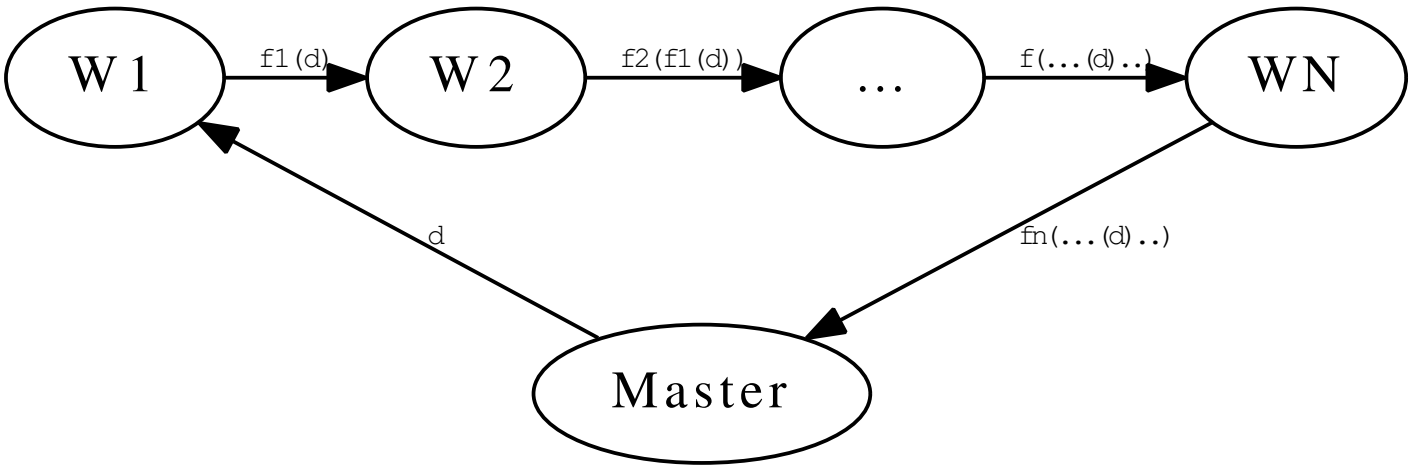
1. Master feeds EOF.

Pipeline

1. Master feeds EOF.
2. Each stage propagates EOF to its successor and exits.

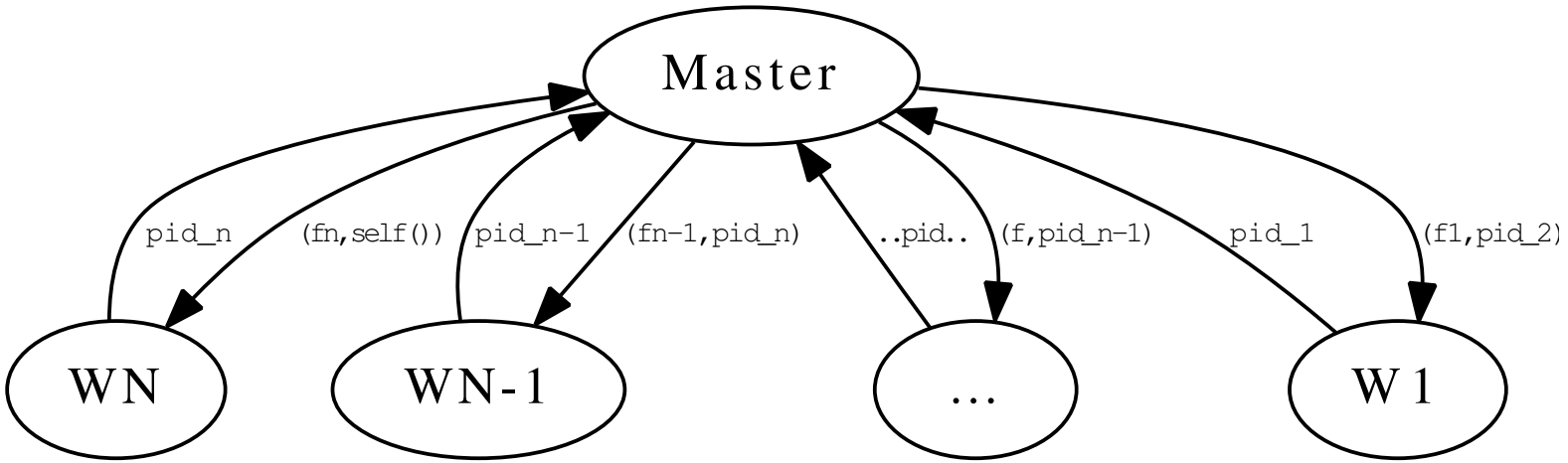
Pipeline

1. Master feeds EOF.
2. Each stage propagates EOF to its successor and exits.



Pipeline

```
1  create(Functions) ->
2     Nodes = get_nodes(length(Functions)),
3     NFS = zip(Nodes, Functions),
4     Pids = foldr(fun(NF, Pids) ->
5                      Pid = make_worker(NF,hd(Pids)),
6                      [Pid] ++ Pids
7                  end, [self()], NFS),
8     Pids.
```



Pipeline

1. Wrapper function that is deployed on the stages.
2. Pattern matching on Data/EOF.
3. Tail-recursive call.
4. Timeout (soft real-time)

```
1  generic_worker(Fun,Pid)  ->
2      Timeout = 60000,%ms
3      receive
4          eof -> Pid ! eof;
5          X ->
6              Pid ! Fun(X),
7              generic_worker(Fun,Pid)
8      after
9          Timeout -> Pid ! eof
10     end.
```

Pipeline

```
1  feed(Data, Pid) ->
2      foreach(fun(Elem) ->
3                      Pid ! Elem,
4                      timer:sleep(10000),
5                end, Data),
6      Pid ! eof.
```

Pipeline

```
1  feed(Data, Pid) ->
2      foreach(fun(Elem) ->
3                      Pid ! Elem,
4                      timer:sleep(10000),
5                end, Data),
6      Pid ! eof.
```

```
1  collect() -> collect([]).
2  collect(Acc) ->
3      receive
4          eof ->
5              reverse(Acc);
6          X ->
7              collect([ X | Acc ])
8      end.
```

Pipeline

```
1  feed(Data, Pid) ->
2      foreach(fun(Elem) ->
3                      Pid ! Elem,
4                      timer:sleep(10000),
5                end, Data),
6      Pid ! eof.
```

```
1  collect() -> collect([]).
2  collect(Acc) ->
3      receive
4          eof ->
5              reverse(Acc);
6          X ->
7              collect([ X | Acc ])
8      end.
```

```
1  run(Data, Head) ->
2      feed(Data, Head),
3      collect().
```

pipeline implementation $\sim$ 70 loc

Pipeline

```
1  feed(Data, Pid) ->
2      foreach(fun(Elem) ->
3                      Pid ! Elem,
4                      timer:sleep(10000),
5                end, Data),
6      Pid ! eof.
```

```
1  collect() -> collect([]).
2  collect(Acc) ->
3      receive
4          eof ->
5              reverse(Acc);
6          X ->
7              collect([ X | Acc ])
8      end.
```

```
1  run(Data, Head) ->
2      feed(Data, Head),
3      collect().
```

pipeline implementation $\sim$ 70 loc

Testing/tracing infrastructure $\sim$ 190 loc

Machines Pool

The network of machines is managed by the erlang run-time.

Machines Pool

The network of machines is managed by the erlang run-time.

```
# less .hosts.erlang
'fujim1'.
'fujim2'.
...
'fujim3.cli.di.unipi.it'
```

Machines Pool

The network of machines is managed by the erlang run-time.

```
# less .hosts.erlang
'fujim1'.
'fujim2'.
...
'fujim3.cli.di.unipi.it'

Eshell V5.7.4  (abort with ^G)
(test@fujim1)1> net_adm:world(verbose).
Pinging test@fujim1 -> pong
Pinging test@fujim2 -> pong
Pinging test@fujim3 -> pong
[test@fujim1,test@fujim2,test@fujim3]
```

Machines Pool

The network of machines is managed by the erlang run-time.

```
# less .hosts.erlang
'fujim1'.
'fujim2'.
...
'fujim3.cli.di.unipi.it'

Eshell V5.7.4  (abort with ^G)
(test@fujim1)1> net_adm:world(verbose).
Pinging test@fujim1 -> pong
Pinging test@fujim2 -> pong
Pinging test@fujim3 -> pong
[test@fujim1,test@fujim2,test@fujim3]
```

The `get_nodes(number)` function returns a list of `number` active nodes, if less nodes are available the list is redundant so that consecutive functions are deployed on the same node (*pipeline*).

```
(test@fujim1)3> pipeline:get_nodes(5).
[test@fujim1,test@fujim2,test@fujim2,test@fujim3,test@fujim3]
```

Erlang library has a pool implementation.

Benchmarks

All benchmarks were run with dummy code whose only purpose was to measure CPU/Network performance under different workload.

Benchmarks

All benchmarks were run with dummy code whose only purpose was to measure CPU/Network performance under different workload.

The D value tunes the load on Network transmission.

Benchmarks

All benchmarks were run with dummy code whose only purpose was to measure CPU/Network performance under different workload.

The D value tunes the load on Network transmission.
The C value tunes the CPU workload.

Benchmarks

All benchmarks were run with dummy code whose only purpose was to measure CPU/Network performance under different workload.

The D value tunes the load on Network transmission.
The C value tunes the CPU workload.

$$g \sim \frac{C}{D}$$

Benchmarks

All benchmarks were run with dummy code whose only purpose was to measure CPU/Network performance under different workload.

The D value tunes the load on Network transmission.
The C value tunes the CPU workload.

$$g \sim \frac{C}{D}$$

Hypothesis:
Each node perform the exact same function, on data of the same size. D and C values are independent.

Benchmarks - Inviso

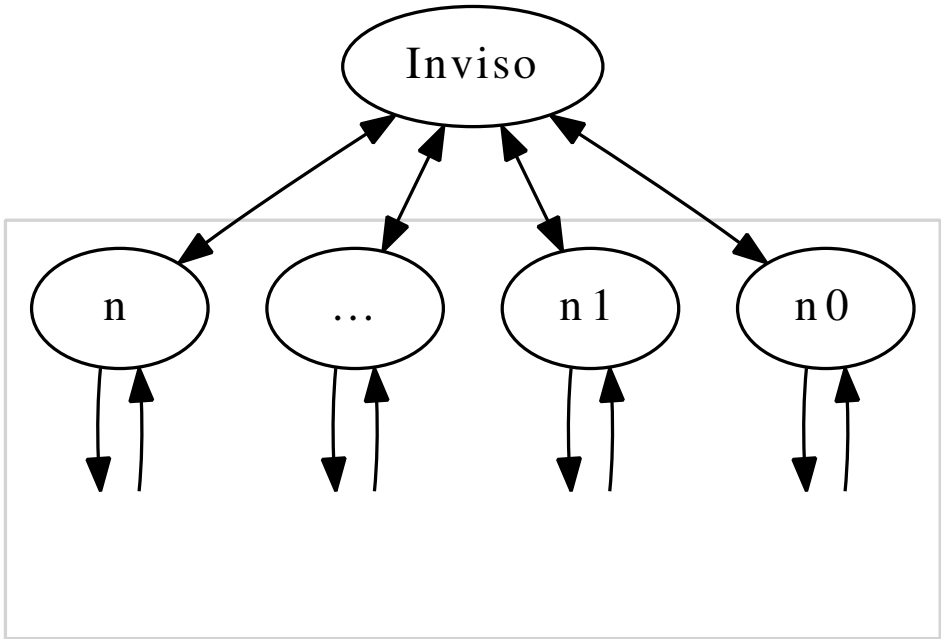
The Inviso framework provided with Erlang was used to trace:

1. function calls, returns
2. messages sent, received

Benchmarks - Inviso

The Inviso framework provided with Erlang was used to trace:

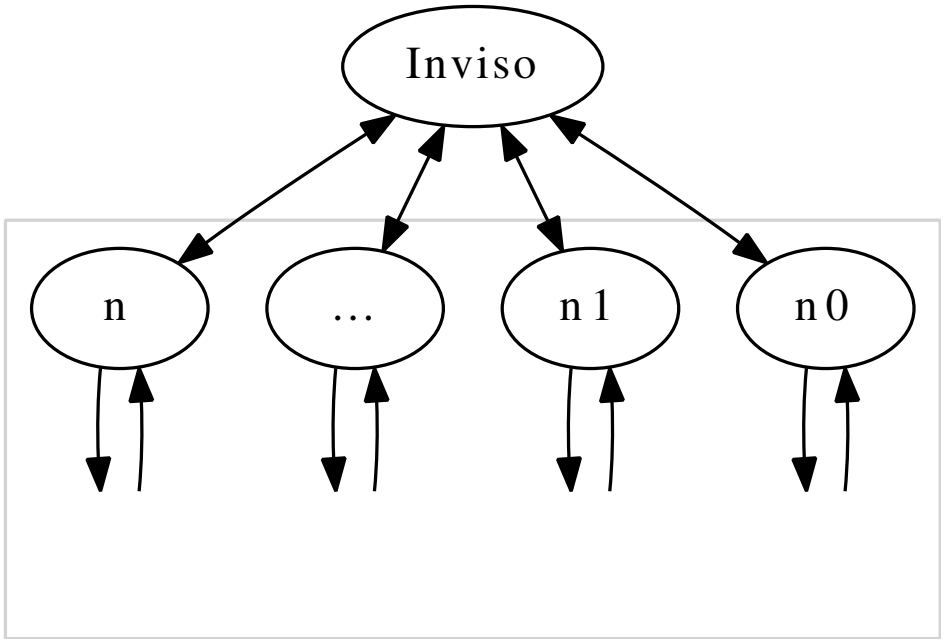
1. function calls, returns
2. messages sent, received



Benchmarks - Inviso

The Inviso framework provided with Erlang was used to trace:

1. function calls, returns
2. messages sent, received

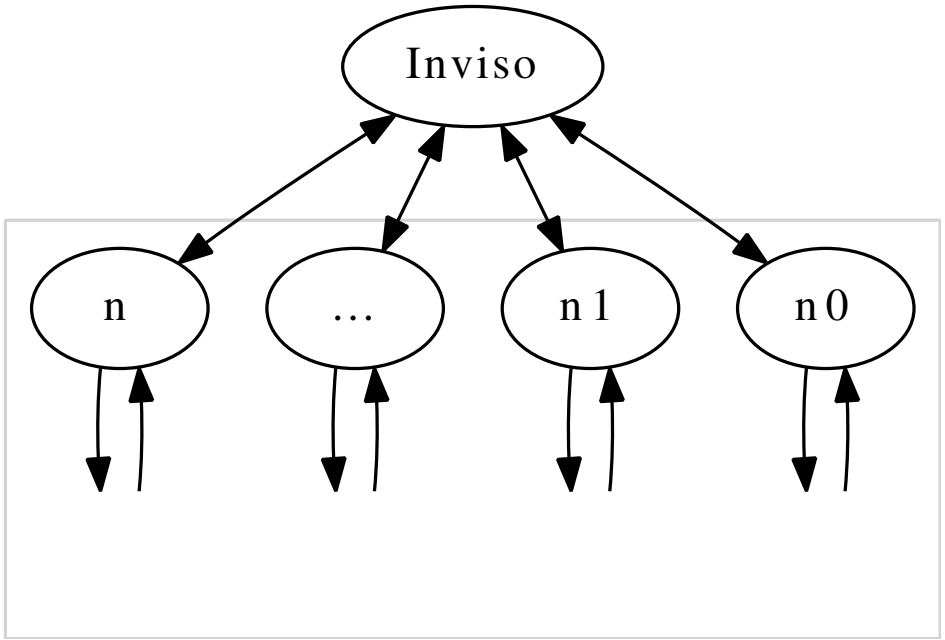


Data, once analyzed, contained precise measure (nanoseconds) of Cpu times and Communication times of every node.

Benchmarks - Inviso

The Inviso framework provided with Erlang was used to trace:

1. function calls, returns
2. messages sent, received

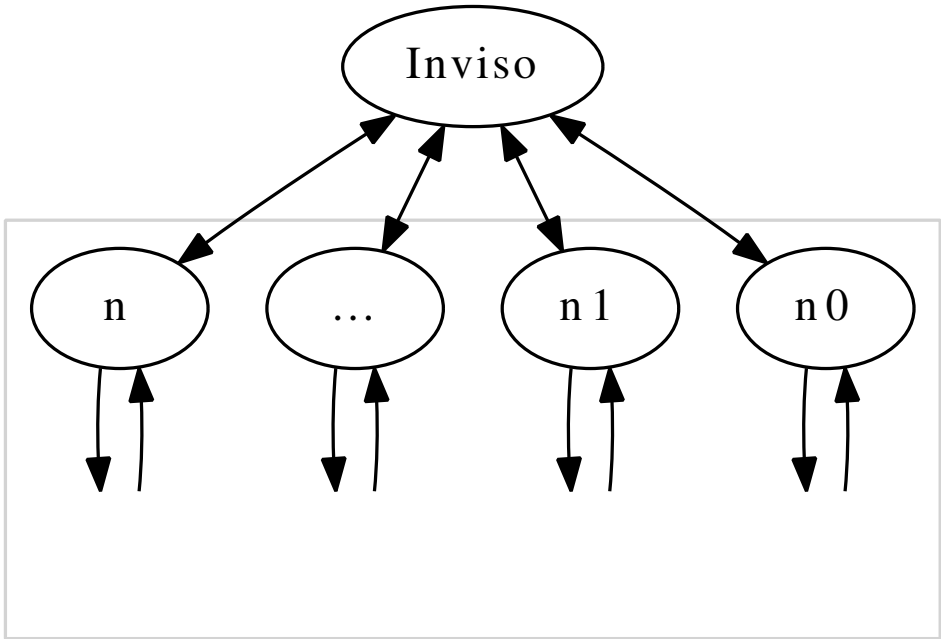


Data, once analyzed, contained precise measure (nanoseconds) of Cpu times and Communication times of every node.
Inviso proved to be very powerfull,

Benchmarks - Inviso

The Inviso framework provided with Erlang was used to trace:

1. function calls, returns
2. messages sent, received

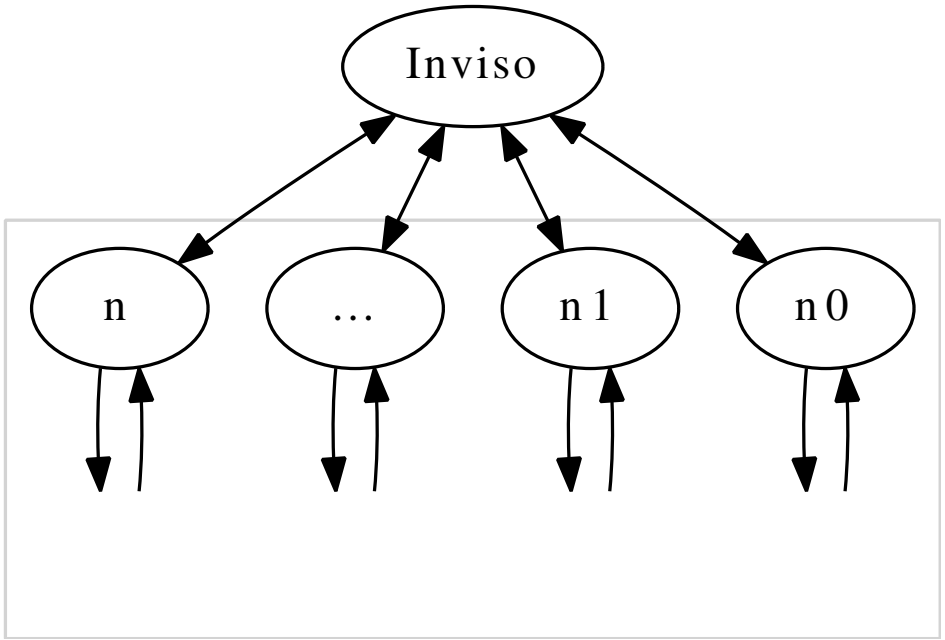


Data, once analyzed, contained precise measure (nanoseconds) of Cpu times and Communication times of every node.
Inviso proved to be very powerfull, very difficult to use,

Benchmarks - Inviso

The Inviso framework provided with Erlang was used to trace:

1. function calls, returns
2. messages sent, received



Data, once analyzed, contained precise measure (nanoseconds) of Cpu times and Communication times of every node.
Inviso proved to be very powerfull, very difficult to use, and eventually very buggy :P

Benchmarks - Inviso

Tell Inviso which node to trace and *how*:

```
1  trace() ->
2    inviso:start(),
3    inviso:add_nodes(Nodes,mytag,[]),
4    TracedList = lists:map(fun(Elem) ->
5         {Elem,[{trace,{relayer,node()}}]} end, OtherNodes)
               ,
6    inviso:init_tracing( TracedList ++ [
7                           {node(),
8                            [{trace,{fun filter/2,[]}}]}
9                            ]),
```

Benchmarks - Inviso

Alternatives

- Log all trace events to file:

```
1 inviso:init_tracing([{client_node(),
2                       [{trace,{file,"client_log"}}]},
3                      {server_node(),
4                       [{trace,{file,"server_log"}}]}])
                            .
```

  they can be later collected and merged.

Benchmarks - Inviso

Alternatives

- Log all trace events to file:

```
1 inviso:init_tracing([{client_node(),
2                       [{trace,{file,"client_log"}}]},
3                      {server_node(),
4                       [{trace,{file,"server_log"}}]}])
                              .
```

  they can be later collected and merged.

- Display all trace events in the shell of the node where they occur:

```
1 inviso:init_tracing([{client_node(),
2                       [{trace,collector}]},
3                      {server_node(),
4                       [{trace,collector}]}]).
```

Benchmarks - Inviso

Tell Inviso what needs to be traced:

```
1   inviso:tpl(OtherNodes,pipeline,funcfunc,'_',
2                              [{'_',[],[{return_trace}]}]),
3   inviso:tf(OtherNodes,all,[send,'receive']),
4   inviso:tf(all,[call,timestamp]).
```

Benchmarks - Inviso

Tell Inviso what needs to be traced:

```
1  inviso:tpl(OtherNodes,pipeline,funcfunc,'_',
2                              [{'_',[],[{return_trace}]}]),
3  inviso:tf(OtherNodes,all,[send,'receive']),
4  inviso:tf(all,[call,timestamp]).
```

alternatives:

- send
- receive
- procs
- call
- return_to
- running : Trace scheduling of processes.
- exiting
- garbage_collection
- timestamp
- cpu_timestamp

Benchmarks - Inviso - filter()

Define a function to treat received data:
Function calls:

```
1  filter(X,CList) ->
2   case X of
3     {trace_ts,Pid,call,{M,F,_A},{MgS,S,McS}} ->
4         Call = {{M,F,Pid},(McS+(S*1000000)+
5                            (MgS*1000000000000))},
6         CList ++ [Call];
7     {trace_ts,Pid,return_from,{M,F,_A},_R,{MgS,S,McS}} ->
8         FTime = (McS+(S*1000000)+
9                            (MgS*1000000000000)),
10        case lists:keysearch({M,F,Pid},1,CList) of
11            {value,{_,STime}} ->
12                ETime = FTime - STime,
13                log({node,Pid,ETime});
14             _ -> ok
15        end,
16        lists:keydelete({M,F,Pid},1,CList);
17 ...
```

Benchmarks - Inviso - filter()

Define a function to treat received data:
Messages:

```
1   {trace_ts,Pid,send,Msg,Dest,{MgS,S,McS}} ->
2       Time = McS+(S*1000000)+(MgS*1000000000000),
3       case Msg of
4           eof -> ok;
5           [[F|_]|_] when is_number(F) ->
6               log({send,Pid,Time});
7           _ -> ok
8       end,
9       CList;
10  ...
```

Benchmarks - Inviso - raw log

Raw log generateb by Inviso:

```
{rec,  "<9434.102.0>", 126857427179982 9}.
{rec,  "<9434.102.0>", 126857427380368 3}.
{rec,  "<9434.102.0>", 126857427580762 7}.
{node, "<9434.102.0>", 19354161}.
{send, "<9434.102.0>", 126857428714750 8}.
{rec,  "<9471.101.0>", 126857428457275 1}.
{node, "<9471.101.0>", 18370486}.
{send, "<9471.101.0>", 126857430294508 3}.
{rec,  "<9472.99.0>",  126857430318489 0}.
{node, "<9434.102.0>", 19671653}.
{send, "<9434.102.0>", 126857430686706 6}.
{rec,  "<9471.101.0>", 126857430429147 4}.
...
```

Benchmarks - Inviso - analyze()

Pass the log through analyze() to extract needed info:

```
1  analyze(Pids,File) when is_list(Pids)->
2      {ok, Log} = file:consult("log.txt"),
3      SendList2 = lists:filter(fun({Type,Pid,Time}) ->
4                               case Type of
5                                   send -> true;
6                                   _ -> false
7                               end
8                       end,Log),
9      {L11,L12,L13} = lists:unzip3(SendList2),
10     SendList= lists:keysort(1,lists:zip(L12,L13)),
11     ...
12     RecList= lists:keysort(1,lists:zip(L22,L23)),
13     ...
14     NodeList= lists:keysort(1,lists:zip(L32,L33)),
15     CPUTimes = ...
16     {ok, FileDescriptor} = file:open(File, [append]),
17     io:format(FileDescriptor, "#service time: ~p~n",
18                               [max(CPUTimes)/1000000]),
19     ...
20     file:close(FileDescriptor).
```

Benchmarks - Inviso - log

Refined log for pmap:

```
{dimData, "nData", nCPU}.
{1500000, "2", 160}.
{node, "<3961.15734.0>", 50637248}.
{node, "<3970.10574.0>", 50419248}.
{total, "time", nmachines}.
{total, "54107364", 2}.
```

Benchmarks - Inviso - log

Refined log for pmap:

```
{dimData, "nData", nCPU}.
{1500000, "2", 160}.
{node, "<3961.15734.0>", 50637248}.
{node, "<3970.10574.0>", 50419248}.
{total, "time", nmachines}.
{total, "54107364", 2}.
```

Refined log for pipeline:

```
#nStages nCPU nData nData
#3 10 50 5
#PIDS: ["<3960.85.0>","<3962.85.0>","<3961.85.0>"]
#service time: 1.915271
1    1.915271    1.892735    1.899414    1.884896    1.848452
2    1.856185    1.842149    1.839809    1.853133    1.85122
3    1.866492    1.848686    1.830319    1.850508    1.842933
```

no network data ;(

Erlang Overhead

Al test were performed with lists of floats on 32bit single-core machines (fujim).
List of N floats $= N * (1 + 4words) * 4bytes = N * 20bytes$.
Erlang introduces an overhead of 20% for data structures.
Overhead due to byte code transfer should be negligible for our applications.

Average Roud Trip Time tested on 10 machines:

Data	Dim	Est Dim	RTT(sec)	MBs
matrix	2000	76 MB	7.876	19
matrix	1000	19 MB	1.895	21
matrix	300	1757 KB	0.184	18
matrix	200	781 KB	0.083	18
matrix	150	439 KB	0.046	18
matrix	100	195 KB	0.020	19
matrix	50	48 KB	0.008	11
lists	1500000	28 MB	3.056	18

Communication Speed $\approx$ 20 MB/sec $\Rightarrow$ 160Mb/s
Very impressive on a 100 Mb/s network :)

Erlang Overhead

Al test were performed with lists of floats on 32bit single-core machines (fujim).
List of N floats $= N * (1 + 4words) * 4bytes = N * 20bytes$.
Erlang introduces an overhead of 20% for data structures.
Overhead due to byte code transfer should be negligible for our applications.

Average Roud Trip Time tested on 10 machines:

Data	Dim	Est Dim	RTT(sec)	MBs
matrix	2000	76 MB	7.876	19
matrix	1000	19 MB	1.895	21
matrix	300	1757 KB	0.184	18
matrix	200	781 KB	0.083	18
matrix	150	439 KB	0.046	18
matrix	100	195 KB	0.020	19
matrix	50	48 KB	0.008	11
lists	1500000	28 MB	3.056	18

Communication Speed $\approx$ 20 MB/sec $\Rightarrow$ 160Mb/s
Very impressive on a 100 Mb/s network :)
Tried with matrix of 2.0 or with diffent values.

Erlang Overhead

Al test were performed with lists of floats on 32bit single-core machines (fujim).
List of N floats $= N * (1 + 4words) * 4bytes = N * 20bytes$.
Erlang introduces an overhead of 20% for data structures.
Overhead due to byte code transfer should be negligible for our applications.

Average Roud Trip Time tested on 10 machines:

Data	Dim	Est Dim	RTT(sec)	MBs
matrix	2000	76 MB	7.876	19
matrix	1000	19 MB	1.895	21
matrix	300	1757 KB	0.184	18
matrix	200	781 KB	0.083	18
matrix	150	439 KB	0.046	18
matrix	100	195 KB	0.020	19
matrix	50	48 KB	0.008	11
lists	1500000	28 MB	3.056	18

Communication Speed $\approx$ 20 MB/sec $\Rightarrow$ 160Mb/s
Very impressive on a 100 Mb/s network :)
Tried with matrix of 2.0 or with diffent values.
No tcpdump/tshark/wireshark so the mistery remains.

Pmap Benchmark

Parallel map was tested with simple lists of floats for finer grained results.

1. a list of D floats is splitted into N sublists.

Pmap Benchmark

Parallel map was tested with simple lists of floats for finer grained results.

1. a list of D floats is splitted into N sublists.
2. on each element is applied C times the erfc function.

Pmap Benchmark

Parallel map was tested with simple lists of floats for finer grained results.

1. a list of D floats is splitted into N sublists.
2. on each element is applied C times the erfc function.
3. the same value received is sent back.

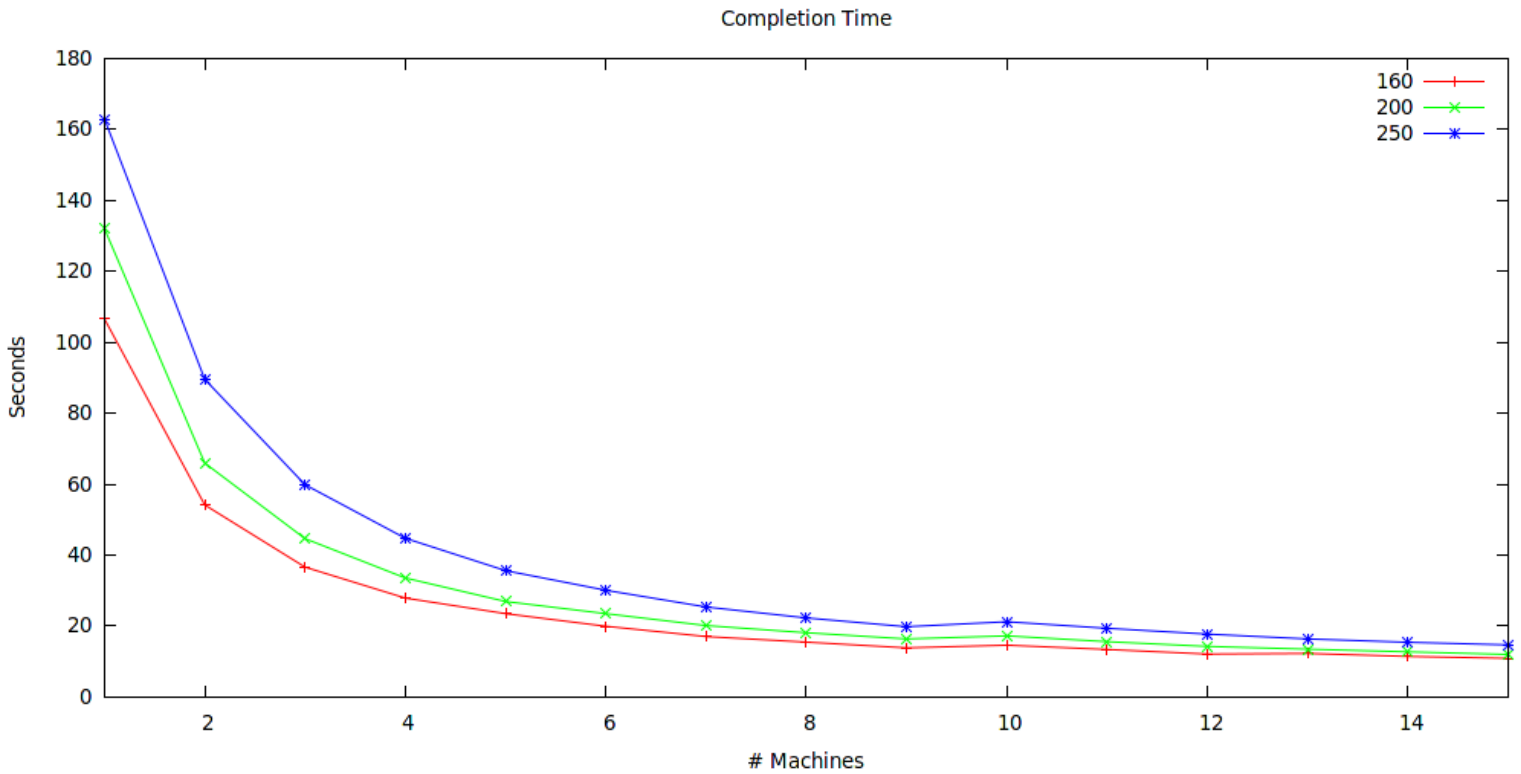
Pmap Benchmark

Parallel map was tested with simple lists of floats for finer grained results.

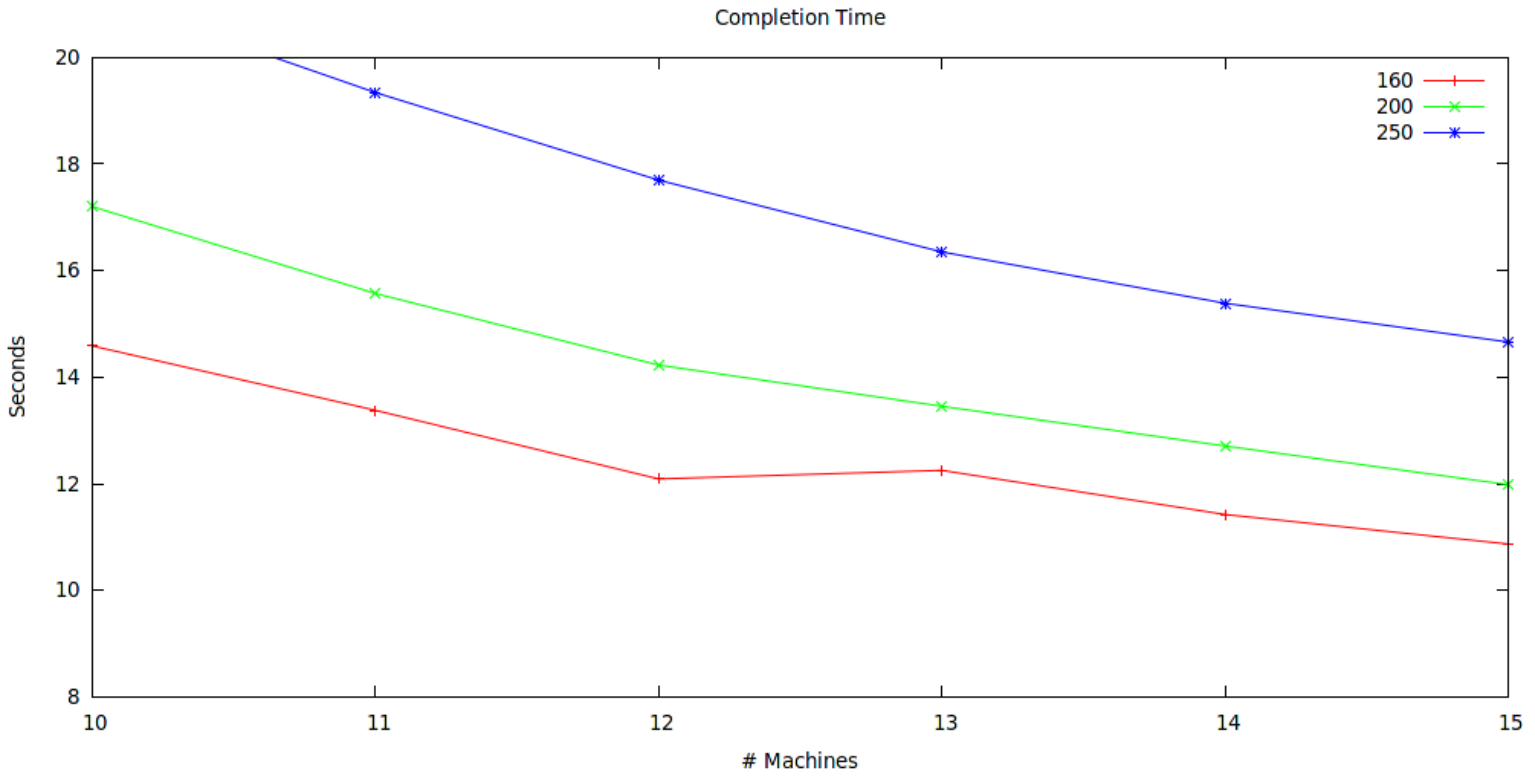
1. a list of D floats is splitted into N sublists.
2. on each element is applied C times the erfc function.
3. the same value received is sent back.

```
1  test(DimData, NData, NCPU) ->
2  ...
3    Data = [list od DimData is splitted to NData]
4    pmap(fun(Vect) ->
5              map(fun(X) ->
6                      foreach(fun(_) ->
7                                   math:erf(X)
8                              end,
9                              seq(1,NCPU)),
10                     X
11                 end, Vect)
12         end, Data),
13 ...
```

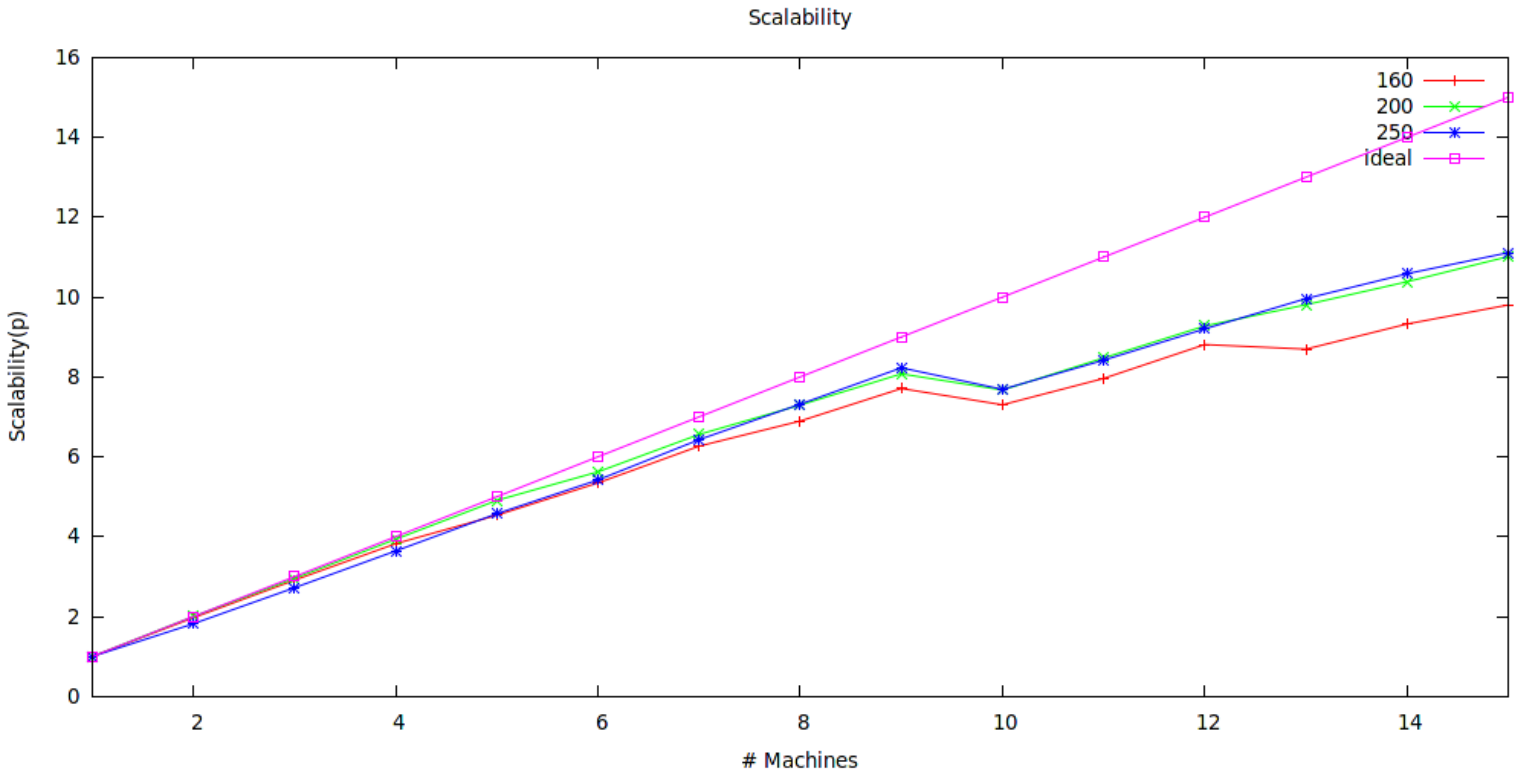
Pmap


Completion Time
Seconds
Machines
160
200
250
0
20
40
60
80
100
120
140
160
180
2
4
6
8
10
12
14

Pmap


Completion Time
Seconds
Machines
160
200
250
20
18
16
14
12
10
8
10
11
12
13
14
15

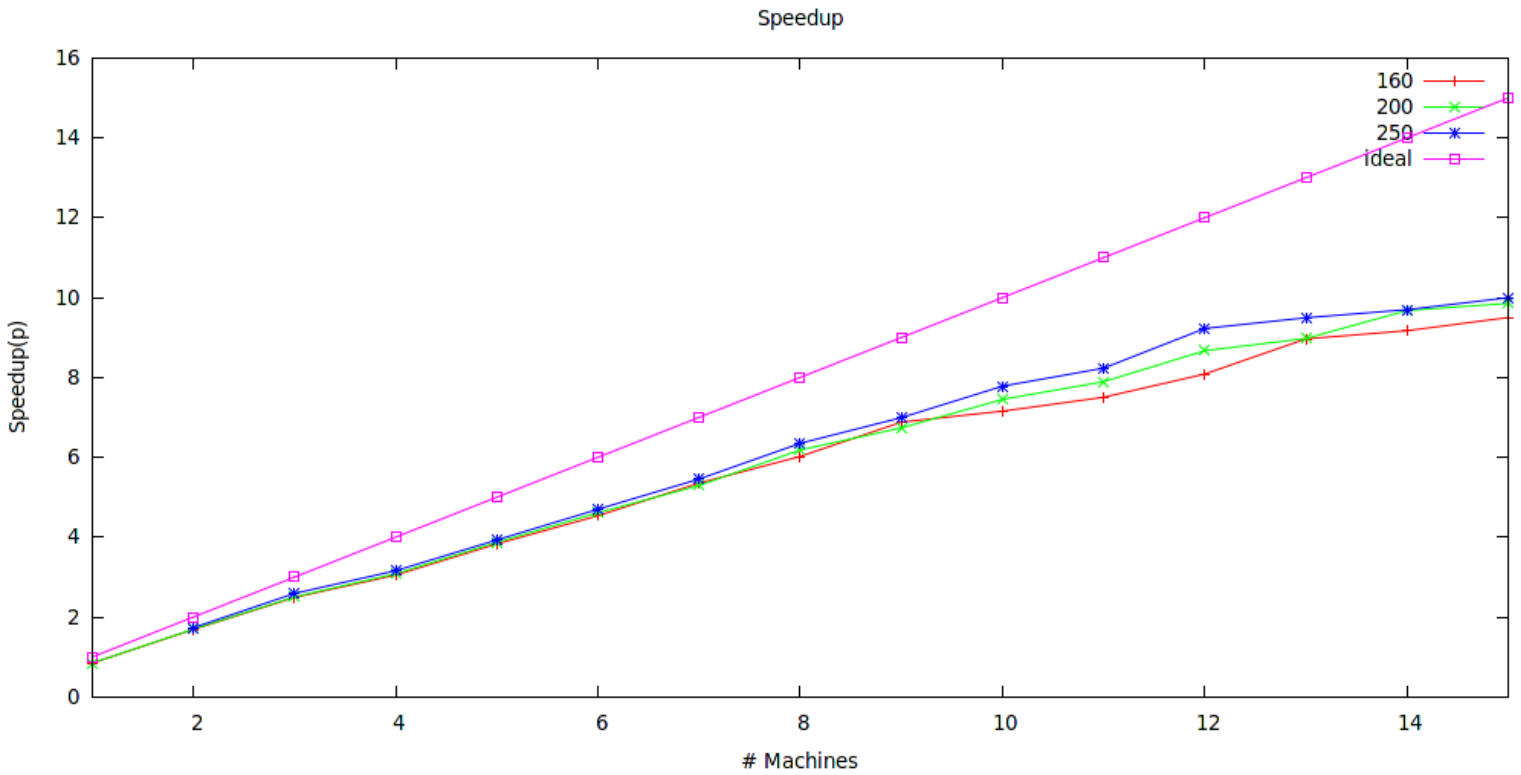
Pmap


Scalability
Scalability(p)
# Machines
160
200
250
ideal
0
2
4
6
8
10
12
14
16

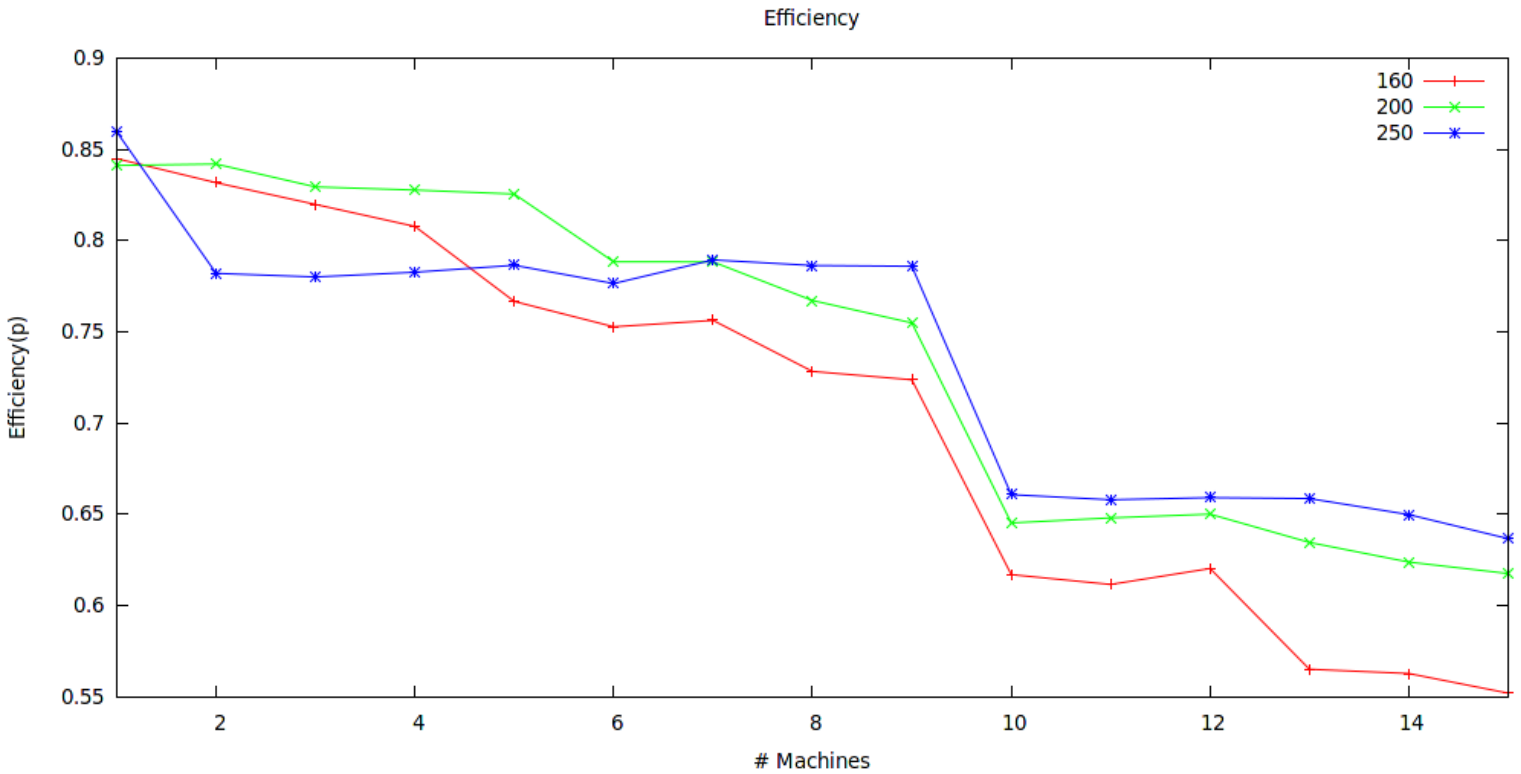
Pmap best sequential

```
1  seque(DimData, NCPU) ->
2     Data = duplicate(DimData,2.0),
3     {Time,Res} =
4         tc(lists,map,
5              [fun(X) ->
6                    foreach(fun(_) ->
7                                  erf(X)
8                            end,
9                            seq(1,NCPU)),
10                   X
11             end, Data]),
12    Time.
```

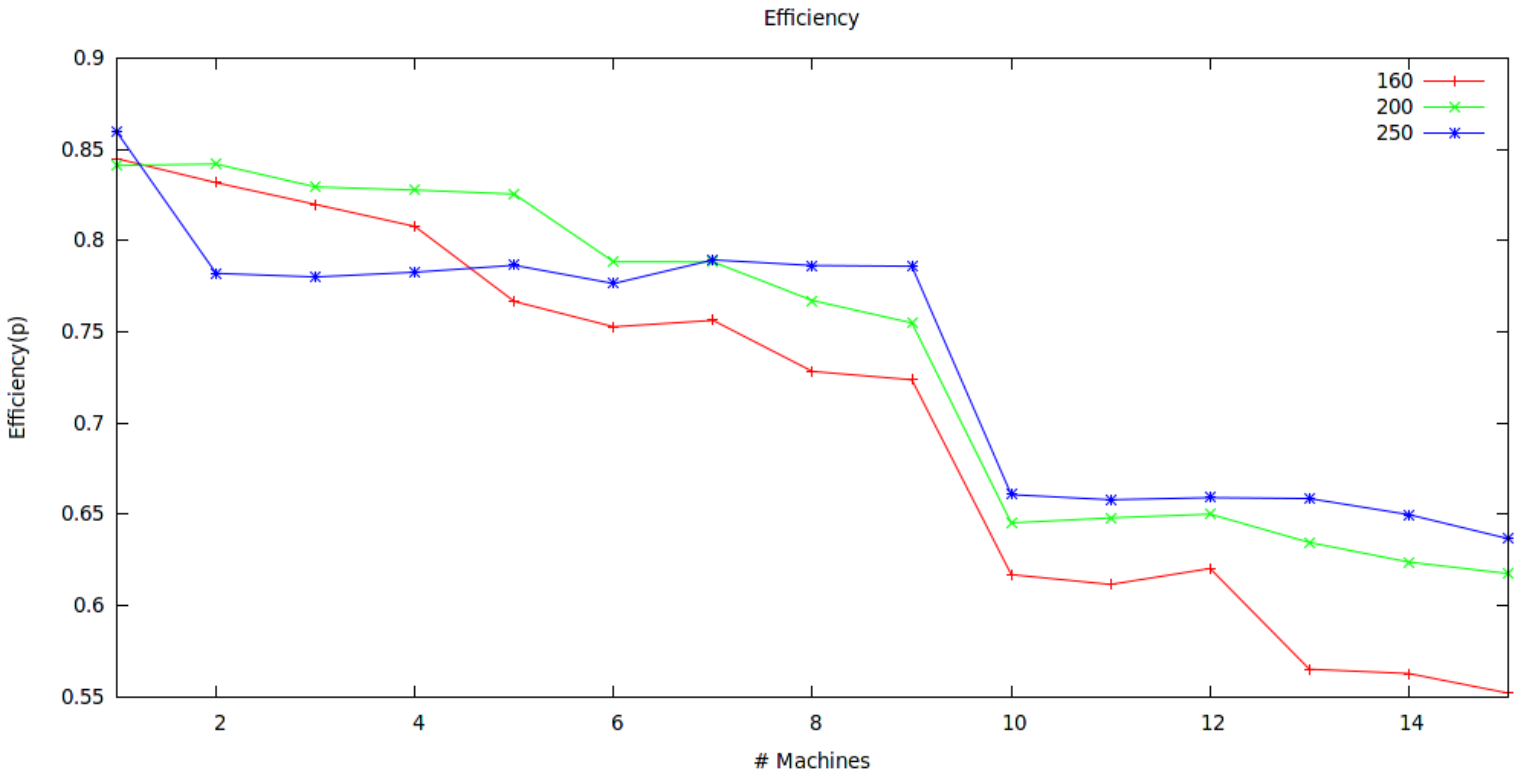
Pmap



Pmap

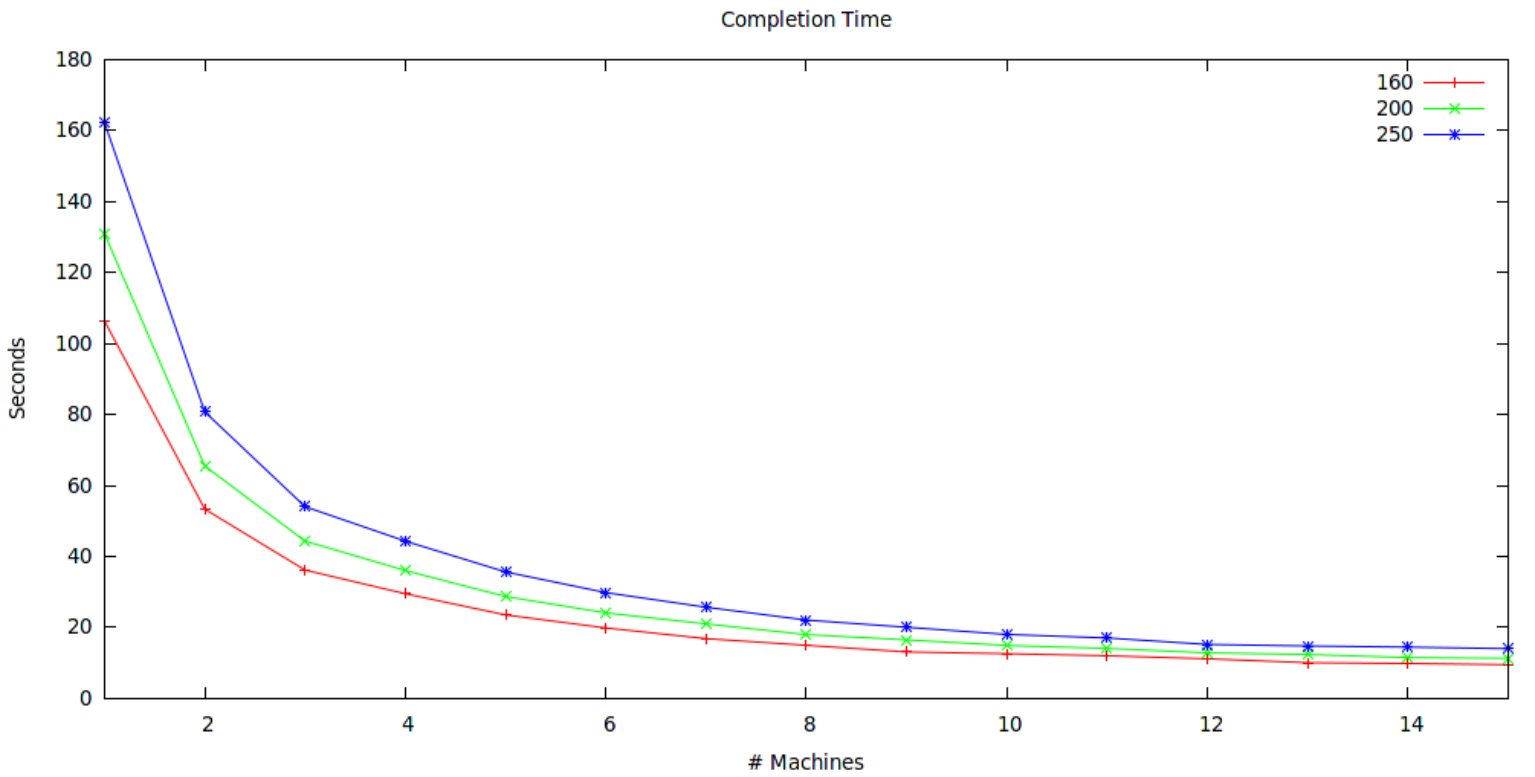

Efficiency
Efficiency(p)
# Machines
160
200
250
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
2
4
6
8
10
12
14

Pmap

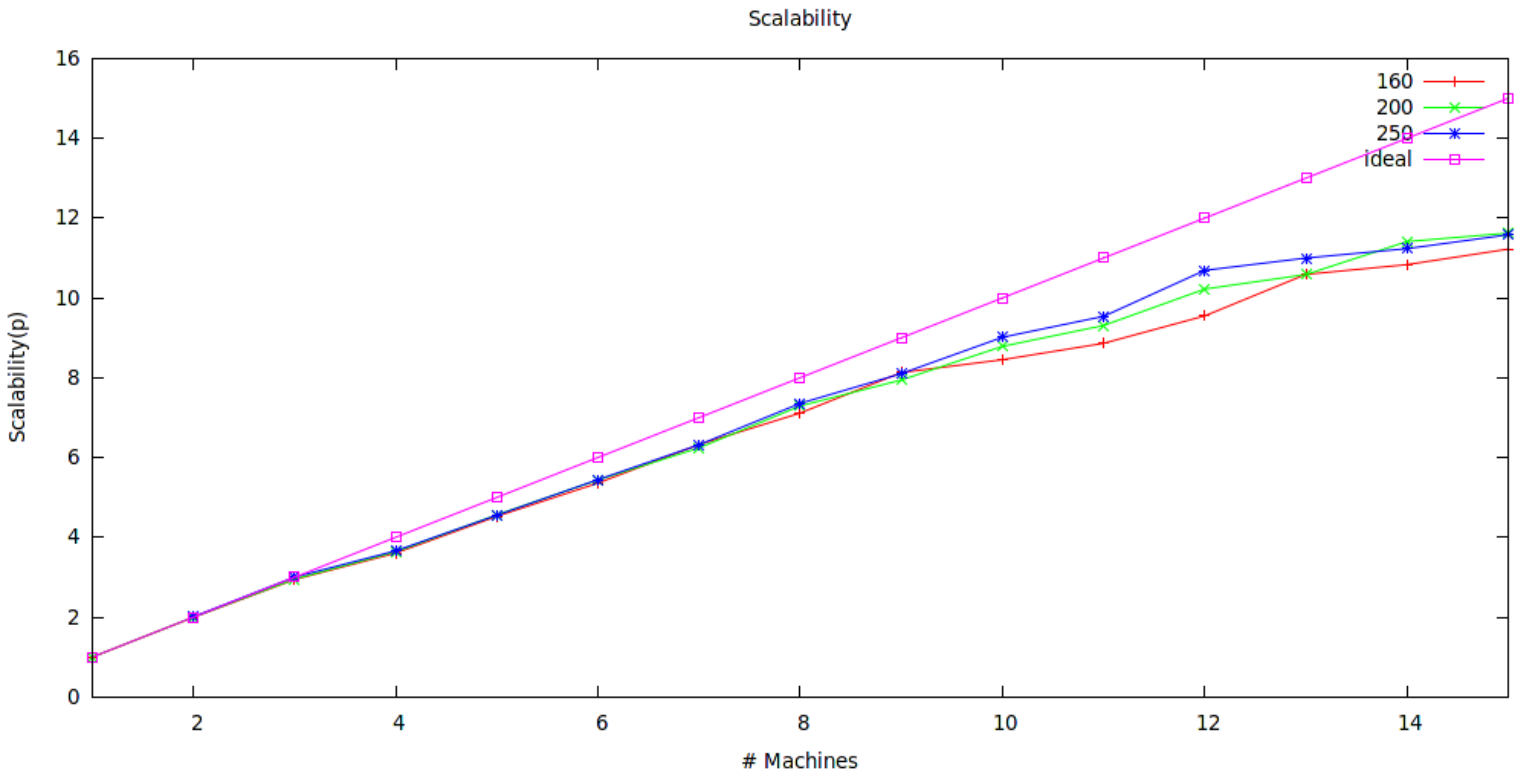


Notice the fall on 10 machines...

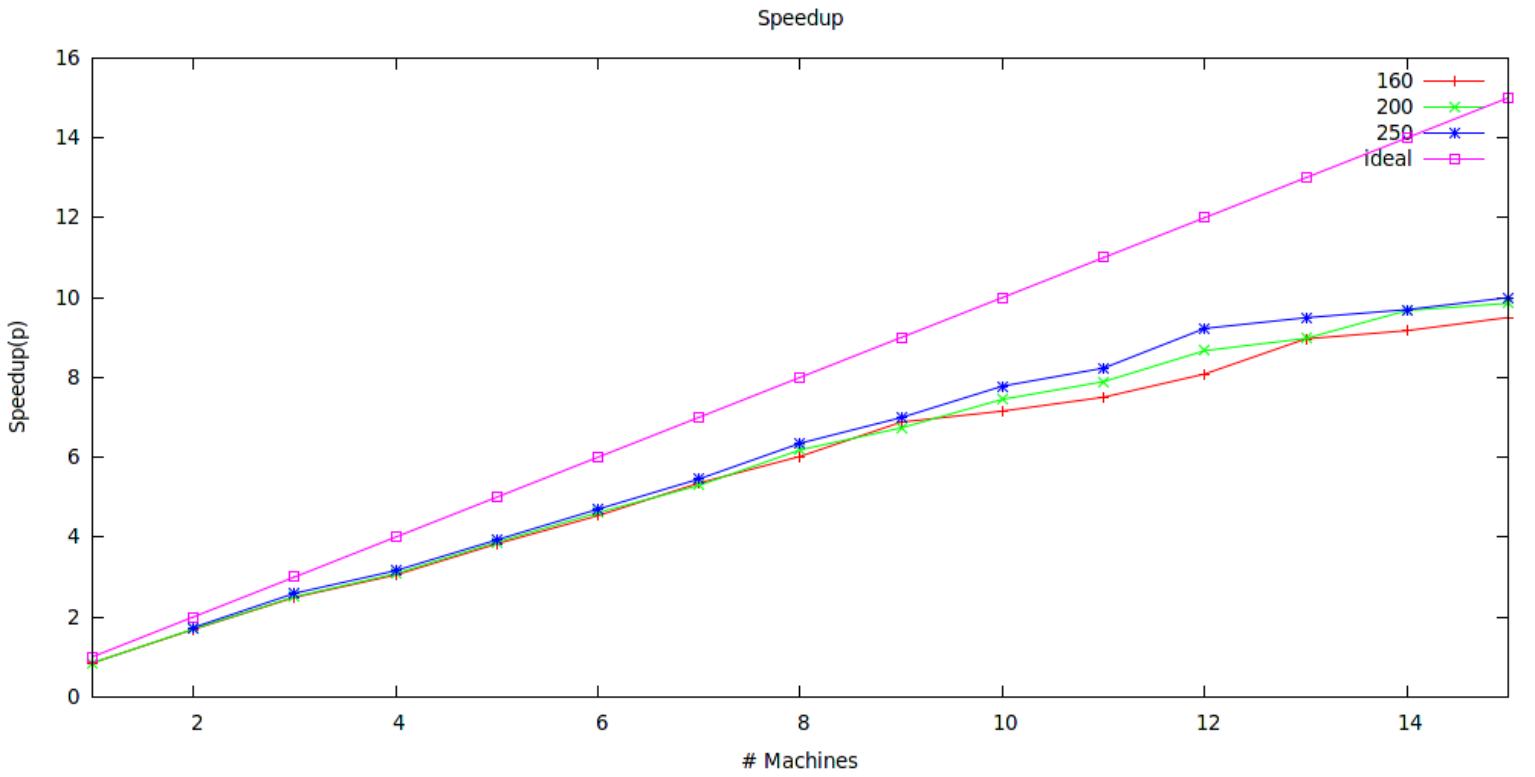
Pmap


Completion Time
Seconds
Machines
160
200
250
0
20
40
60
80
100
120
140
160
180
2
4
6
8
10
12
14

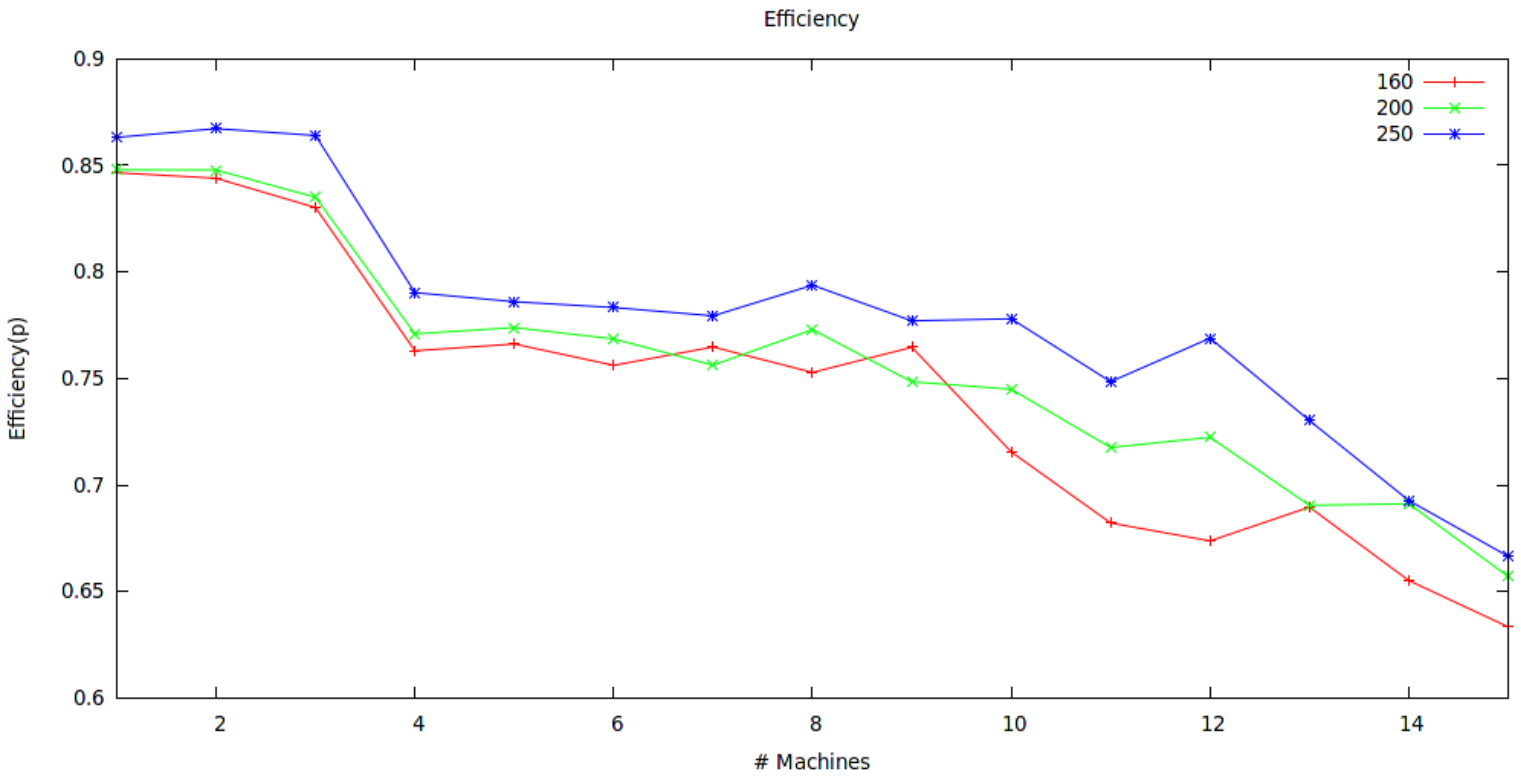
Pmap


Scalability
Scalability(p)
# Machines
160
200
250
Ideal
0
2
4
6
8
10
12
14
16

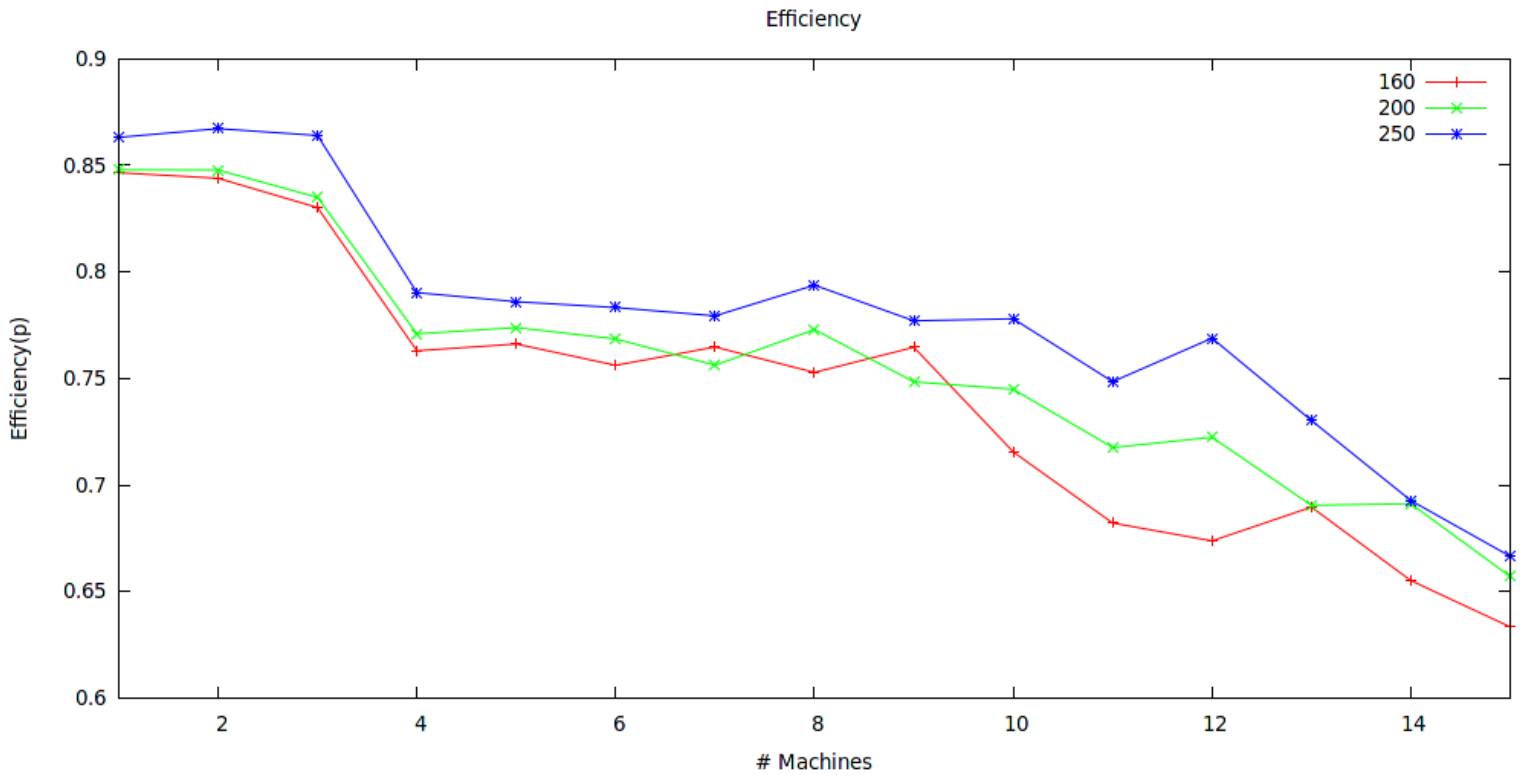
Pmap


Speedup
Speedup(p)
# Machines
160
200
250
Ideal
0
2
4
6
8
10
12
14
16

Pmap


Efficiency
Efficiency(p)
# Machines
160
200
250
0.9
0.85
0.8
0.75
0.7
0.65
0.6
2
4
6
8
10
12
14

Pmap



Much better :)

Pipeline Benchmark

A small library to handle matrix multiplication was implemented:

1. generate 50x50 matrix with the same value 2.0.: `duplicate2(50,2.0)`

Pipeline Benchmark

A small library to handle matrix multiplication was implemented:

1. generate 50x50 matrix with the same value 2.0.: `duplicate2(50,2.0)`
2. multiply a list of matrices: `multiply_matrix([M1,M2,...,Mn])`

Pipeline Benchmark

A small library to handle matrix multiplication was implemented:

1. generate 50x50 matrix with the same value 2.0.: `duplicate2(50,2.0)`
2. multiply a list of matrices: `multiply_matrix([M1,M2,...,Mn])`
3. C power: `multiply_matrix(duplicate(C,duplicate2(D,X)))`

Pipeline Benchmark

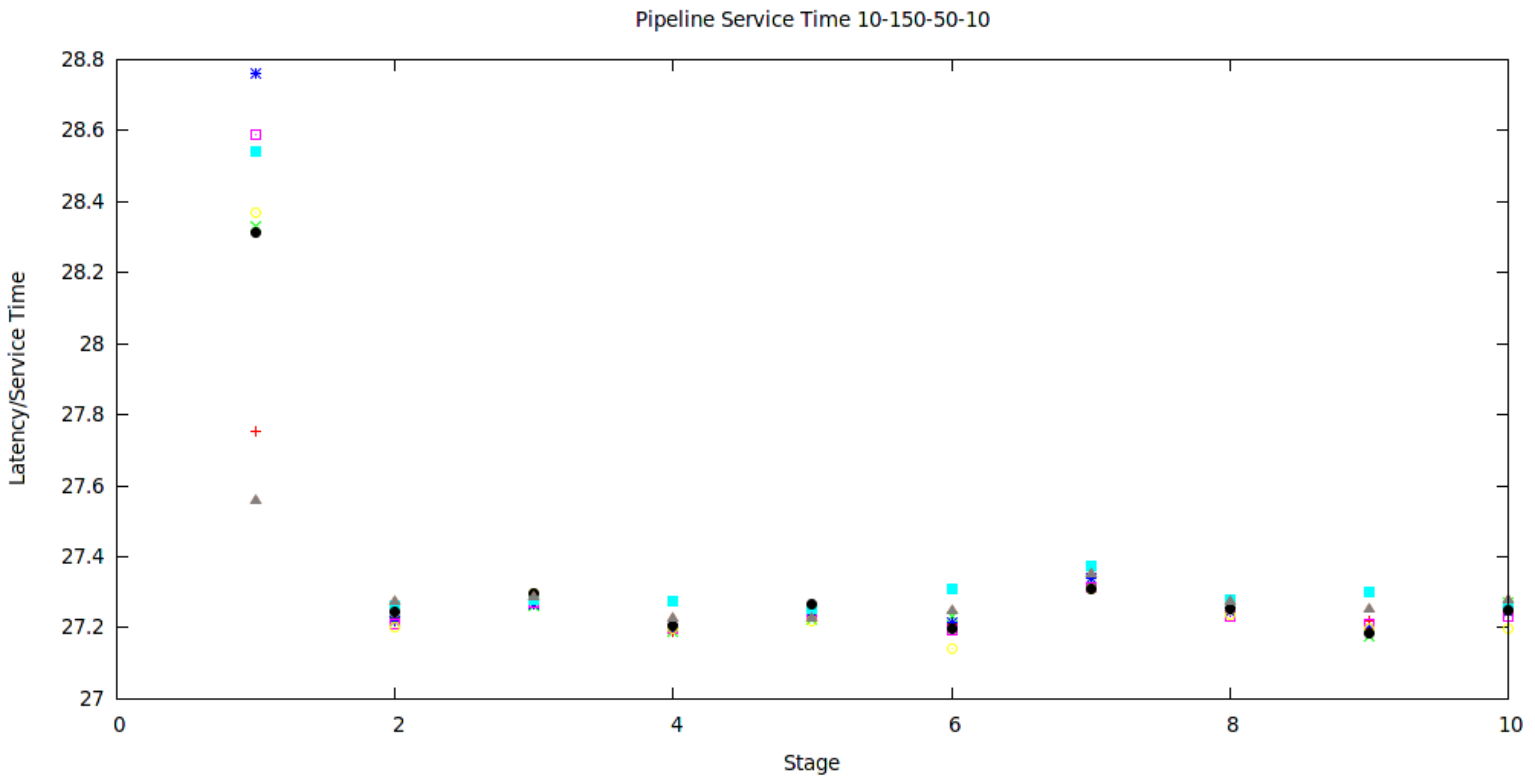
A small library to handle matrix multiplication was implemented:

1. generate 50x50 matrix with the same value 2.0.: `duplicate2(50,2.0)`
2. multiply a list of matrices: `multiply_matrix([M1,M2,...,Mn])`
3. C power: `multiply_matrix(duplicate(C,duplicate2(D,X)))`

```
1  test(NStages,NCPU,DData,NData) ->
2  ...
3    Pids = create(duplicate(NStages,
4               fun(X) ->
5                 Matrix = duplicate2(50,2.0),
6                 multiply_matrix(duplicate(NCPU,Matrix)),
7                 X
8               end)),
9  ...
10   Res = run(duplicate(NData, duplicate2(DData,2.0)),
11              hd(Pids)),
12 ...
```

Pipeline

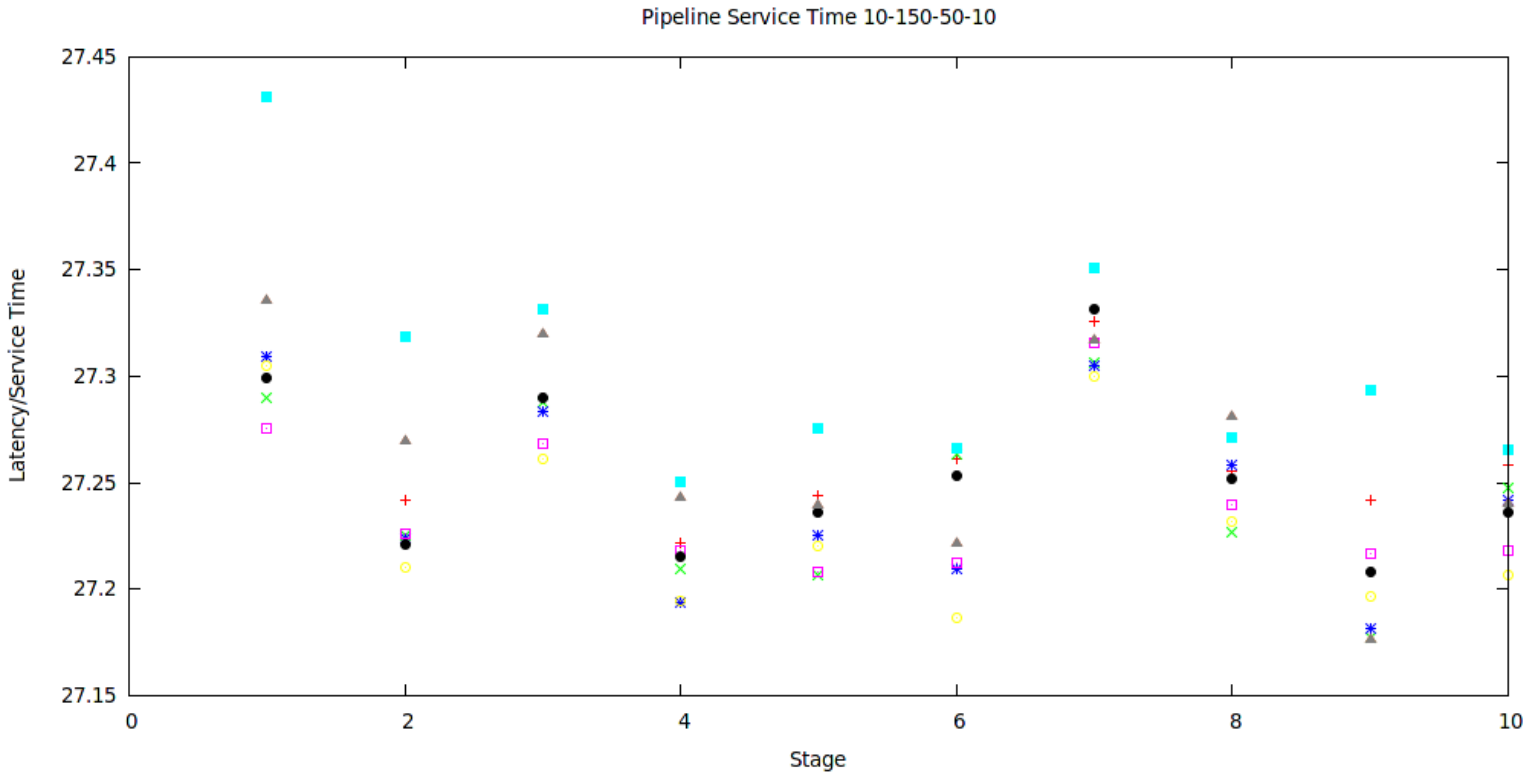
Feeding is too fast, in this case 2 sec.
Inter departure time $T_P \ll T_S$



Ts_α: 28.244
C=150 D=50

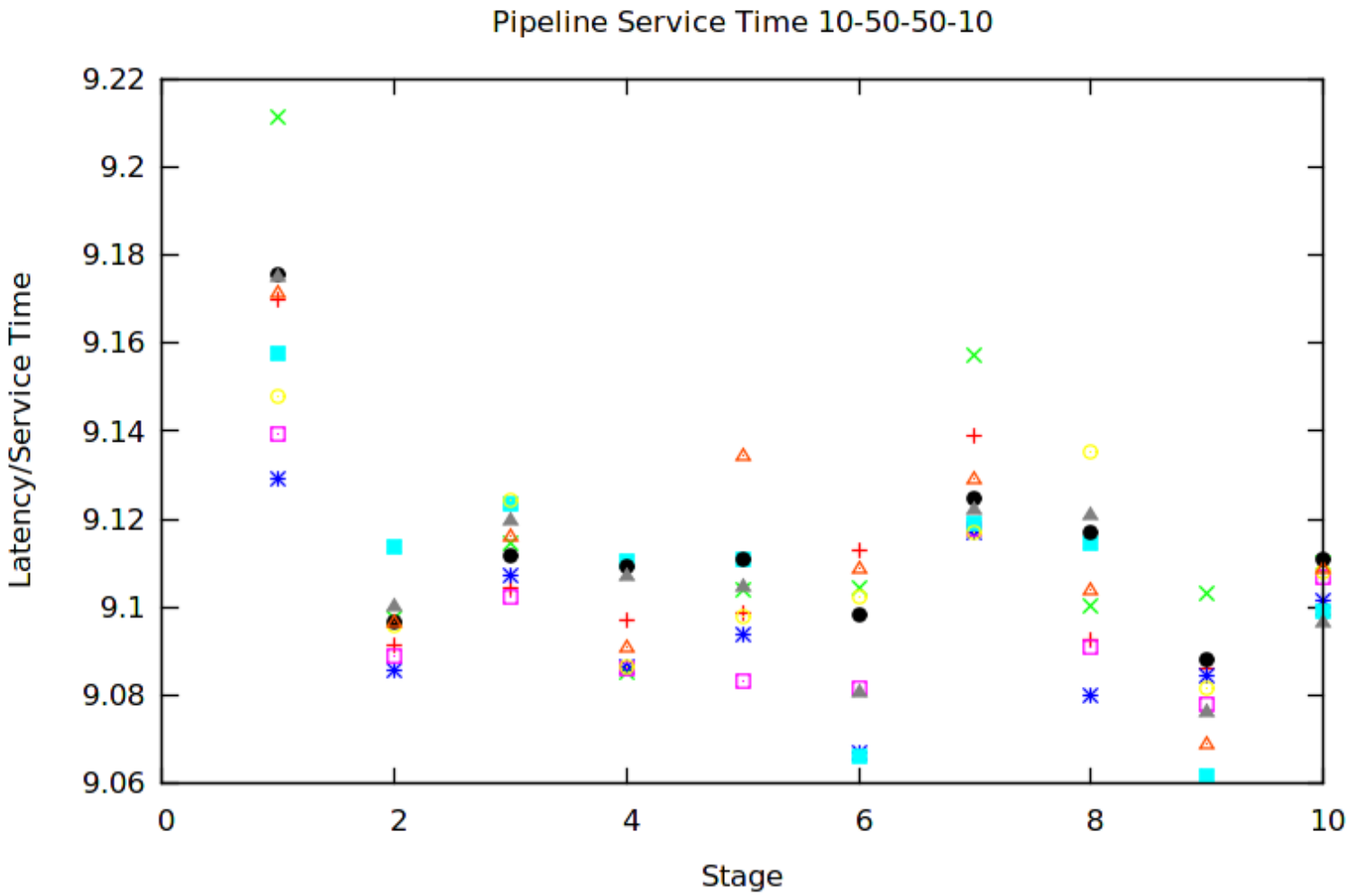
Pipeline Benchmark: head is the bottleneck

Feeding is close to average Service Time: 27 sec.



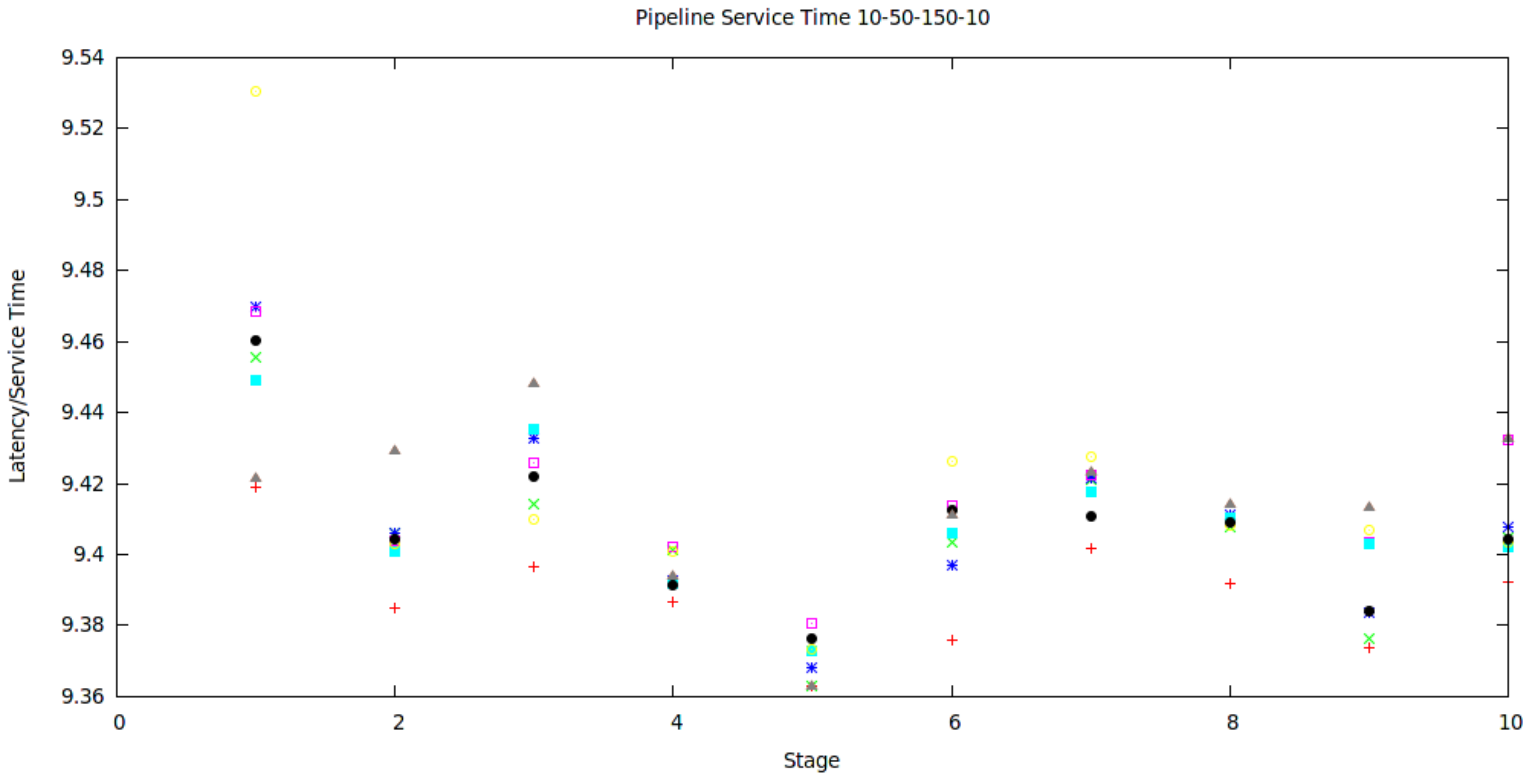
Ts_α: 27.374
C=150 D=50

Pipeline



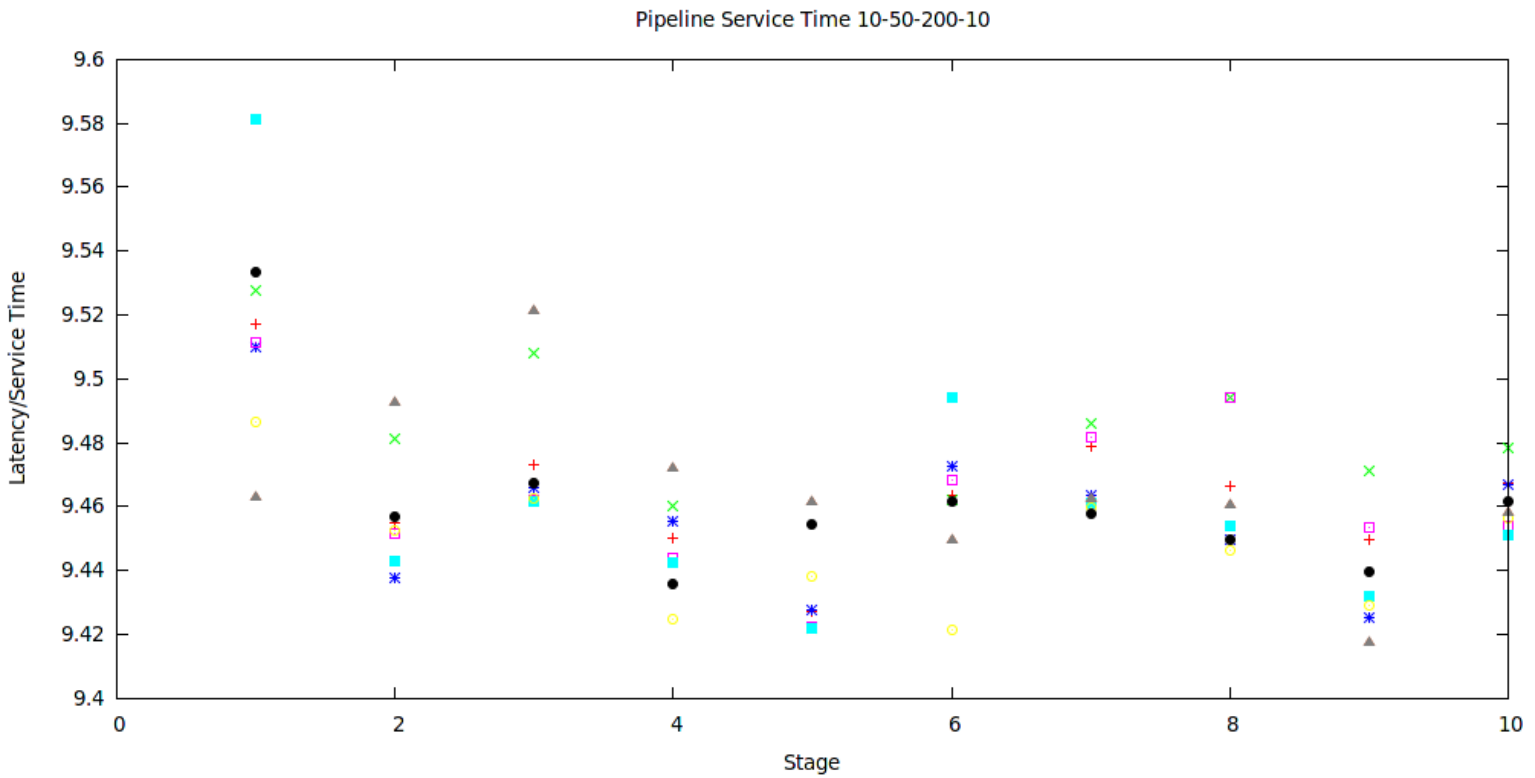
C=50 D=50

Pipeline - Ts_i



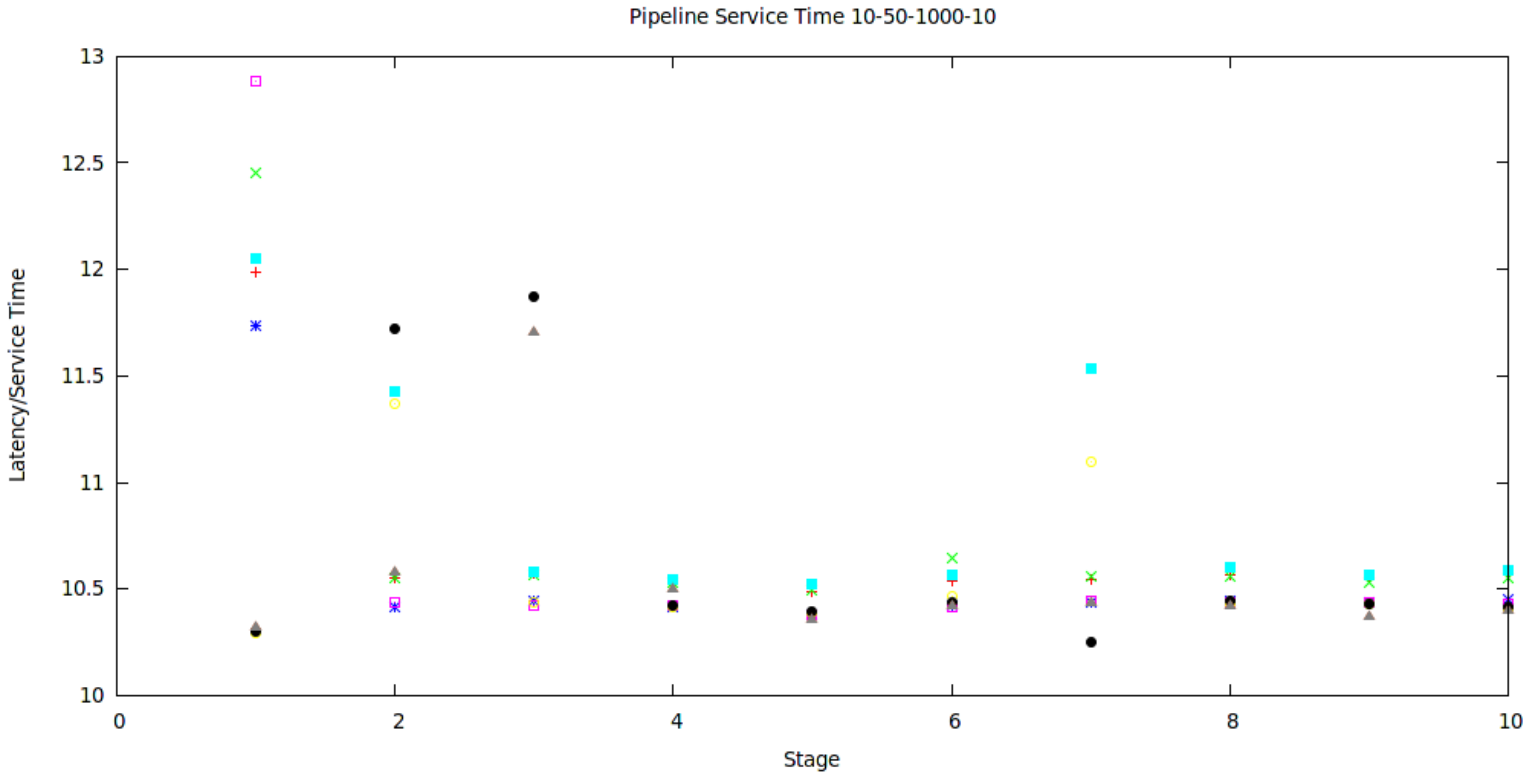
C=50 D=150

Pipeline - Ts_i



C=50 D=200

Pipeline - Ts_i



C=50 D=1000

Pipeline - Looking for the unbalanced case

$T_A = \text{average}(\text{InterArrivalTimes})$
$T_S = max\{Ts_i\}$

C	D	T_A	T_S	ρ
50	50	9.199	9.211453	1.00
50	150	9.629	9.530386	0.98
50	200	9.868	9.581177	0.97
50	300	10.069	9.799503	0.97
50	1000	19.452	12.882535	0.66

matrix of 1000 floats $\sim$ 19MB $\Rightarrow$ RTT: 1.895 sec.

Going further

- implement other skeletons:
  - farm as a general case for pmap
  - fold as a particular case for pipeline
- exploit higher order
  - compose skeletons: `pmap(fun(X) -> pmap(X) end, [[1..3000],[1..2000]])`
- do things the right way: erlang pool manager
- fault tolerance: handle exceptions
- learn from errors: autonomic first-stage pipeline
- make *ports* for dusty deck code

Going further

- implement other skeletons:
  - farm as a general case for pmap
  - fold as a particular case for pipeline
- exploit higher order
  - compose skeletons: `pmap(fun(X) -> pmap(X) end, [[1..3000],[1..2000]])`
- do things the right way: erlang pool manager
- fault tolerance: handle exceptions
- learn from errors: autonomic first-stage pipeline
- make *ports* for dusty deck code
- (lot further) make a manager with inviso/heartbeat...

Going further

Erlang as a didactic/specification language:

- high level components: pmap $\rightarrow$ pipeline $\rightarrow$ farm
  - write application specification with full abstraction

Going further

Erlang as a didactic/specification language:

- high level components: pmap $\rightarrow$ pipeline $\rightarrow$ farm
  - write application specification with full abstraction
- middle level: my little framework
  - write framework with erlang abstraction (send/receive/spawn...)

Going further

Erlang as a didactic/specification language:

- high level components: pmap $\rightarrow$ pipeline $\rightarrow$ farm
  - write application specification with full abstraction
- middle level: my little framework
  - write framework with erlang abstraction (send/receive/spawn...)
- TCP/UDP, ssh, posix ...
  - write framework with posix abstraction

Going further

Erlang as a didactic/specification language:

- high level components: pmap $\rightarrow$ pipeline $\rightarrow$ farm
  - write application specification with full abstraction
- middle level: my little framework
  - write framework with erlang abstraction (send/receive/spawn...)
- TCP/UDP, ssh, posix ...
  - write framework with posix abstraction
  - only short-coming is no shared memory support