A Performance Evaluation of Elixir

Filipe Varjão
Federal University of Pernambuco
Recife-Pernambuco-Brazil

UFPE

Languages

- ± 690 programming languages in the world by wikipedia

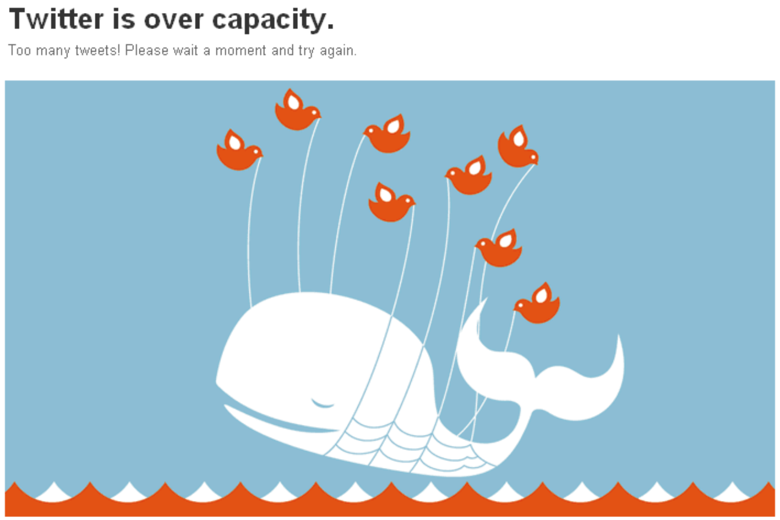
What is worng ?

- Massively scalable
- High availability
- Concurrency
- Fault tolarance

...

Examples

- Twitter: first version writing in Ruby on Rails changed to Scala



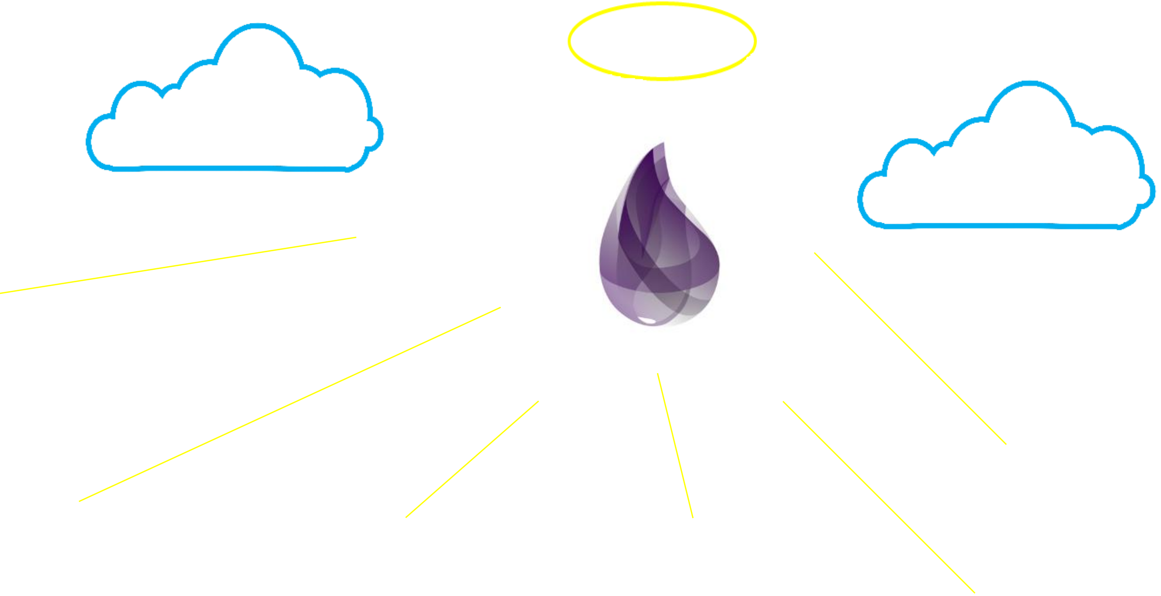
- Facebook: Writing in PHP changed the backend to new compiler in C++ call HipHop and chat to Erlang

Solution ?

Real Problem

- The most languages were not designed to withstand massive concurrency
- Languages based on a shared memory model do not support the necessary scalability
- Functional languages such Erlang or Haskel are hard to use

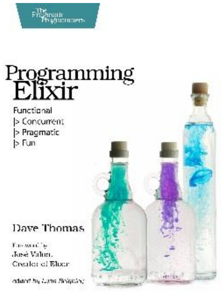
Elixir

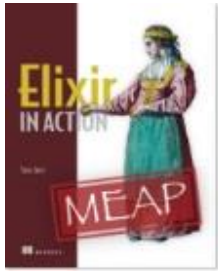


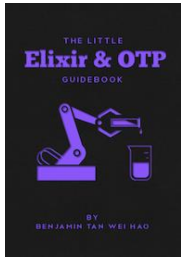
- Has friendly syntax (Productivity)
- Concurrency-Oriented language
- Distributed
- Fault-Tolerant (making the system stay up despite failure)
- Elixir can use Erlang libraries and can call Erlang code (Compatibility)

Elixir

- Created by Jose Valim in 2011
- Elixir debuts on the TIOBE index at position #212 in 2012
- Programming Elixir (Dave Thomas, foreword José Valim) 2013
- Elixir in Action (Saša Jurić) 2013
- Introducing Elixir (Simon St. Laurent, J. David Eisenberg) 2013
- The Little Elixir & OTP guigdebook (Benjamin Tan Wei Hao) 2014
- ElixirConf |> 2014







Elixir the language

- Easy to learn and understand such Ruby
- Message passing
- Pattern matching
- Metaprogramming via macros
- List comprehensions

...

What do we need ?

- Building backend systems for big applications where the massive concurrency and tolorency-failure are required

What do we need ?

- Building backend systems for big applications where the massive concurrency and tolorency-failure are required

How do we want ?

- With clear code and productivity

What do we need ?

- Building backend systems for big applications where the massive concurrency and tolorency-failure are required

How do we want ?

- With clear code and productivity

How can we do it ?

- Using Elixir

Intel MPI Benchmark

“Intel® MPI Benchmarks performs a set of MPI performance measurements for point-to-point and global communication operation for a range of message sizes.“

- The main goal is measure the efficiency of latency and throughput.

https://software.intel.com/sites/products/documentation/hpc/ics/imb/32/IMB_Users_Guide/IMB_Users_Guide.pdf

Intel MPI Benchmark

- Single Transfer: only exchange one message between two processes
- Parallel Transfer: one message exchange per pair of processes, but several pairs communicate in parallel
- Collective Transfer: measure MPI collective operations

Intel MPI Benchmark

Single Transfer	Parallel Transfer	Collective
PingPong	SendRecv	Bcast / Multi-Bcast
PingPongSpecificSource	Exchange	Allgather / Multi-Allgather
PingPing	Multi-PingPong	Allgatherv / Multi-Allagatherv
PingPingSpecificSource	Mult-PingPing	Alltoall / Multi-Alltoall
	Multi-Sendrecv	Alltoallv / Multi-Alltoallv
	Multi-Exchange	Scatter / Multi-Scatter
		Scatterv / Multi-Scatterv
		Gather / Multi-Gather
		Gatherv / Multi-Gatherv
		Reduce / Multi-Reduce
		Reduce_scatter / Multi-Reduce_scatter
		Allreduce / Multi-Allreduce
		Barrier / Multi-Barrier

PingPing

This benchmark measures the efficiency in the treatment of blocking, which happens whenever a process receives a message at the same time it sends another one.

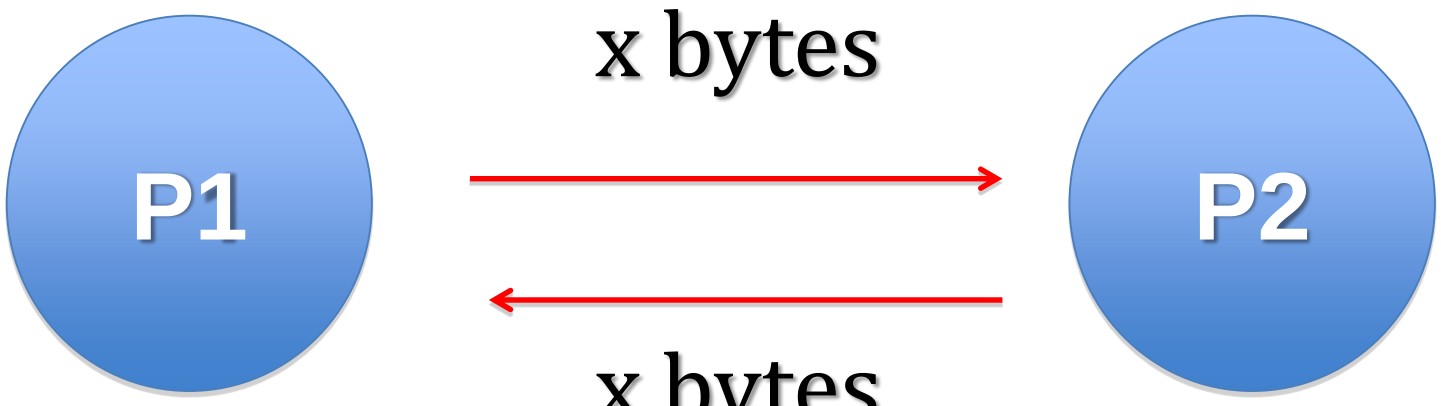
It also registers the process latency time and throunghput the system offers (processing each message)

PingPong

Similar to PingPing, but the message is obstructed by incoming messages.

Intel MPI Benchmark

- PingPing: Asynchronous message passing between two processes.
- PingPong: Synchronous message passing between two processes.

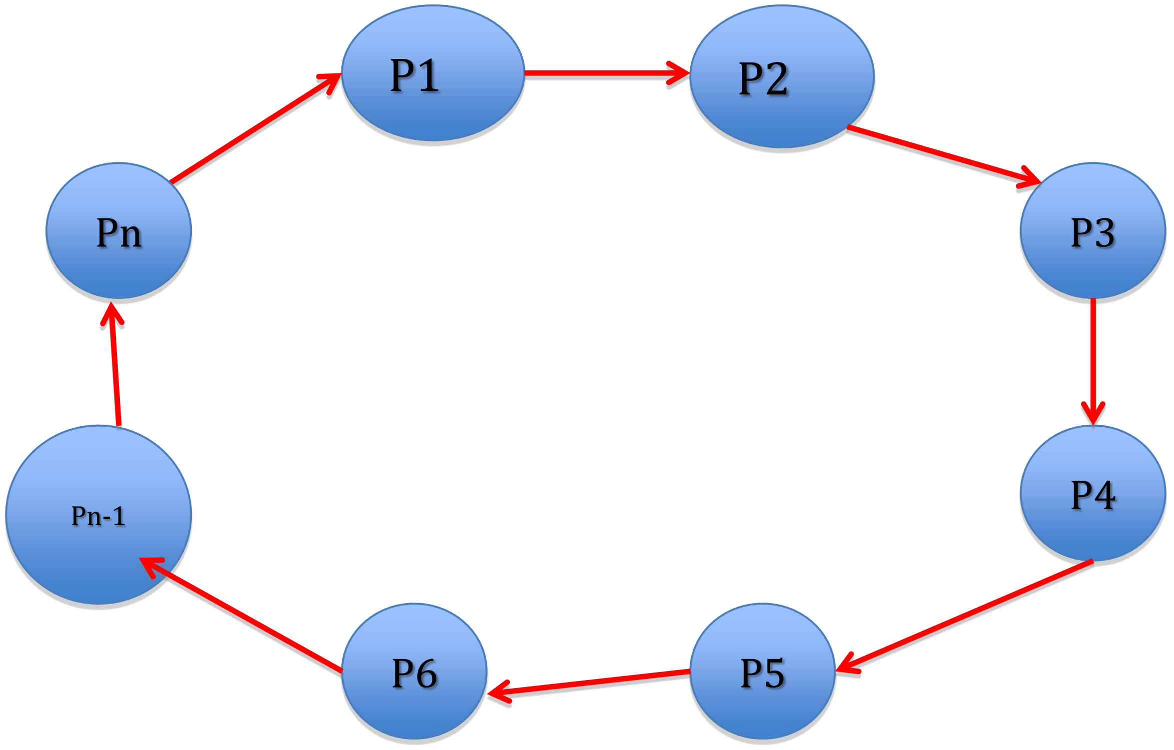


SendRecv

- Thread ring send-receive is the simplest test of parallel transfer
- Many process are created
- Each one sends to the right and receive from the left neighbor in the chain
- Two process will report the bi-directional of the system, as obtained by the optimized function

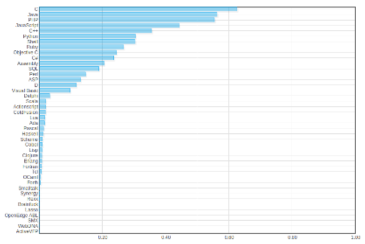
SendRecv

- Massive creation of processes for parallel transfer test



Languages

Language Rank	Types	Spectrum Ranking
1. Java		100.0
2. C		99.2
3. C++		95.5
4. Python		93.4
5. C#		92.2
6. PHP		84.6
7. Javascript		84.3
8. Ruby		78.6
9. R		74.0
10. MATLAB		72.6



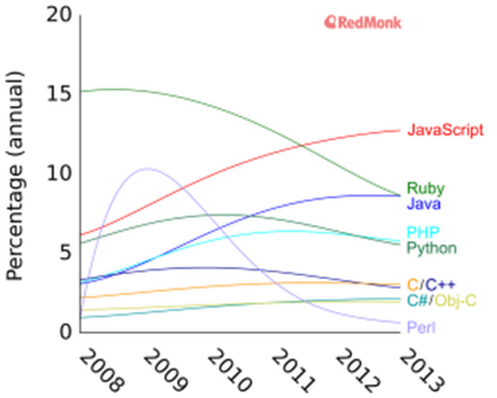
Jul 2014	Jul 2013	Change	Programming Language	Ratings	Change
1	1		C	17.145%	-0.48%
2	2		Java	15.688%	-0.22%
3	3		Objective-C	10.294%	+0.05%
4	4		C++	5.520%	-3.23%
5	7	↑	(Visual) Basic	4.341%	+0.01%
6	6		C#	4.051%	-2.16%
7	5	↓	PHP	2.916%	-4.27%
8	8		Python	2.656%	-1.38%
9	10	↑	JavaScript	1.806%	-0.04%
10	12	↑	Transact-SQL	1.759%	+0.19%
11	9	↓	Perl	1.627%	-0.52%
12	13	↑	Visual Basic .NET	1.495%	+0.24%
13	37	⇈	F#	1.093%	+0.86%
14	11	↓	Ruby	1.072%	-0.51%
15	45	⇈	ActionScript	1.067%	+0.86%
16	-	⇈	Swift	1.054%	+1.05%
17	17		Delphi/Object Pascal	1.031%	+0.34%
18	15	↓	Lisp	0.829%	-0.04%
19	18	↓	MATLAB	0.781%	+0.10%
20	20		Assembly	0.777%	+0.20%

IEEE SPECTRUM

TIOBE SOFTWARE THE CODING STANDARDS COMPANY

stackoverflow

New GitHub repositories

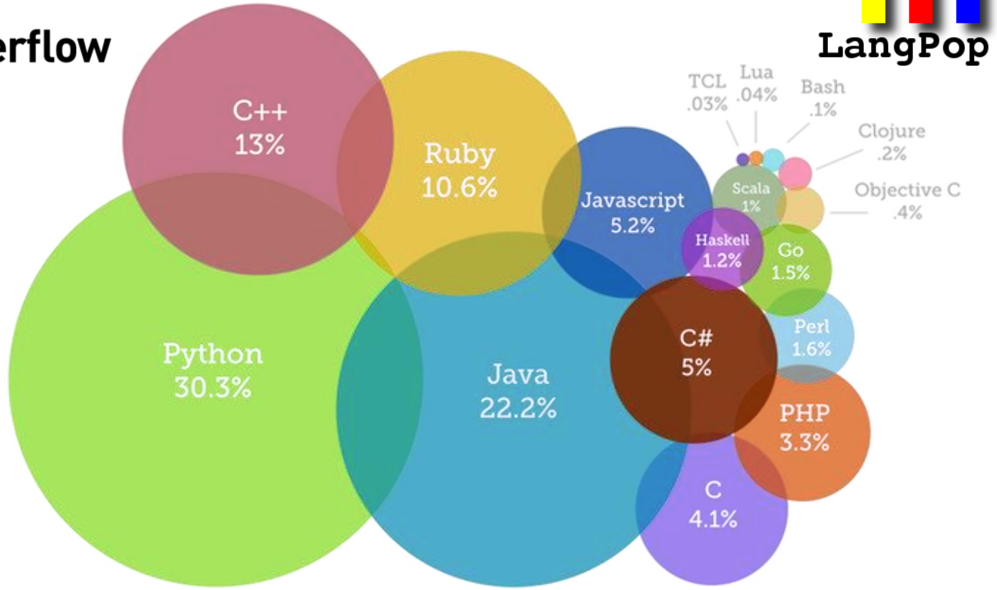


Trends over time, smoothed to make it a little easier to follow

Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities

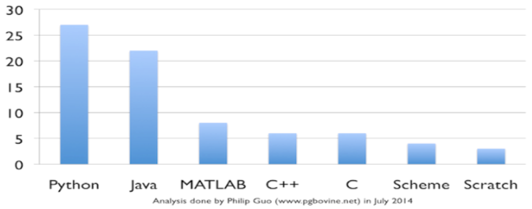
Most Popular Coding Languages of 2014

LangPop.com



Most Popular Programming Languages of 2014

GitHub



Intel MPI Benchmark

- Erlang
- Elixir
- Java
- Scala
- Python
- Ruby
- ooErlang

Environment Configuration

Operating System	Ubuntu Server 12.10 64bits
Hardware	Intel i7-3770@3.4Ghz
Programming languages	Elixir v0.12.4 Java Oracle Version - 1.8.0-b132 x 64 Erlang R16B03 (erts-5.10.4) x64 ooErlang 1.0 Scala version 2.11.0. RC3 Python 2.7.3 Ruby 1.8.7

- ooErlang is a conservative meta-programming object-oriented extension for Erlang
- objects are introduced with a syntax close to Java, making it easier to adopt by object-oriented programmers.

https://sites.google.com/site/ooerlang1/

https://github.com/jucimarjr/ooerlang

PingPing

```
def run(size, r) do

   data = generate_data(size)
   spawnStart = time_microseg()

   parent = self()
   p1 = spawn(fn -> pingping(data, parent, r) end)
   p2 = spawn(fn -> pingping(data, parent, r) end)

   spawnEnd = time_microseg()
   timeStart = time_microseg()
   send(p1, {:init, self, p2})
   send(p2, {:init, self, p1})
   finalize(p1)
   finalize(p2)
   timeEnd = time_microseg()
   totalTime = timeEnd - timeStart
   spawnTime = spawnEnd - spawnStart
end
```

PingPing

```
def pingping(_, pid, 0), do: send(pid ,{:finish, self})

def pingping(data, pid, r) do
  receive do
    {:init, ^pid, peer} ->
      send(peer, {self, data})
      pingping(data, pid, r - 1)
    {peer, data} ->
      send(peer, {self, data})
      pingping(data, pid, r - 1)
  end
end

def finalize(p1) do
  receive do
    {:finish, ^p1} ->
      :ok
  end
end
```

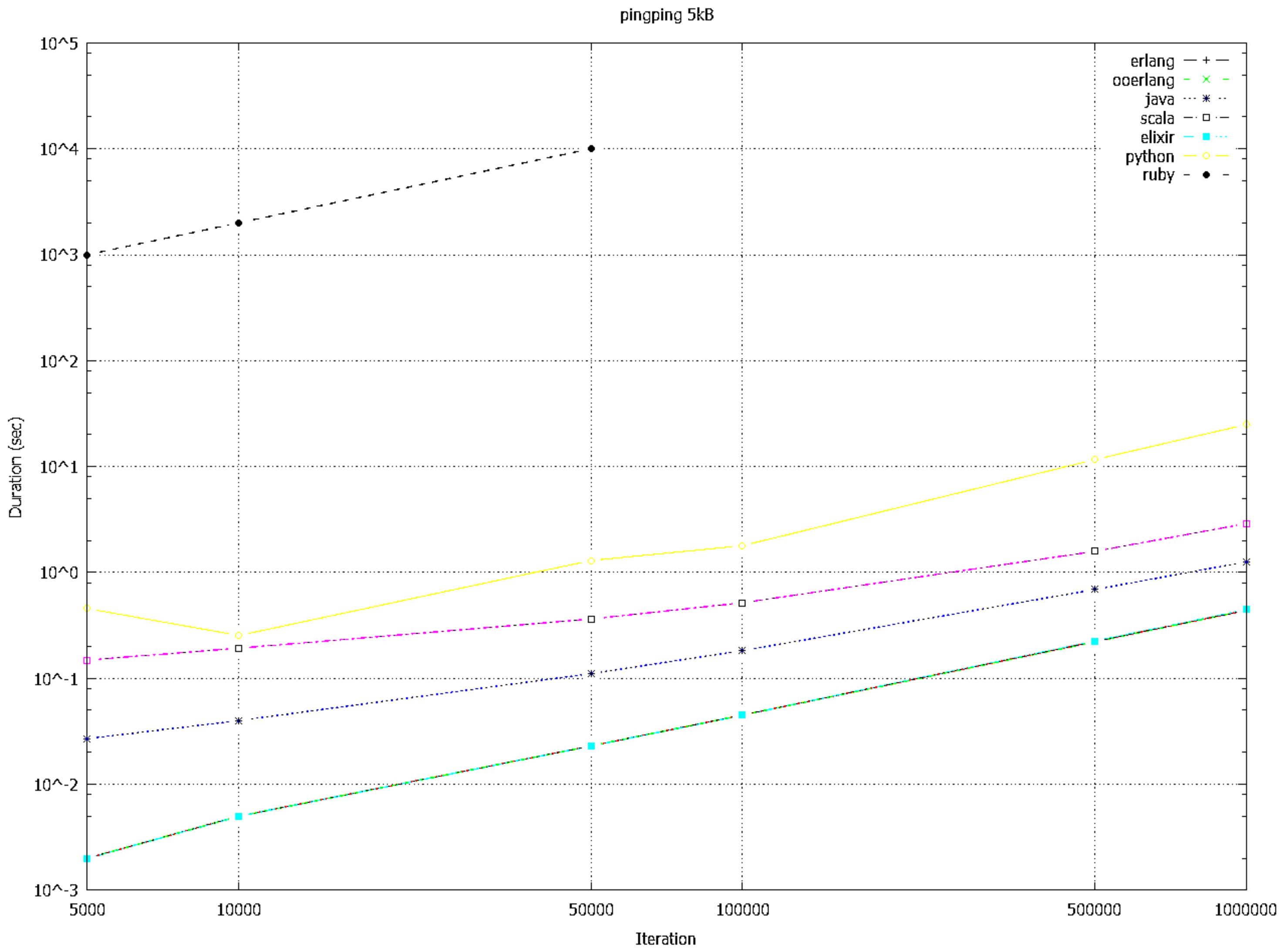
PingPing

```
def bandwidth_calc(data, time) do
  megabytes = (:erlang.size(data) / :math.pow(2, 20))
  seconds = time * 1.0e-6
  megabytes / seconds
end

def generate_data(size), do: generate_data(size, [])

def generate_data(0, bytes), do: :erlang.list_to_binary(bytes)

def generate_data(size, bytes), do: generate_data(size - 1, [1 | bytes])

def time_microseg() do
  {ms, s, us} = :erlang.now()
  (ms * 1.0e+12) + (s * 1.0e+6) + us
end
```

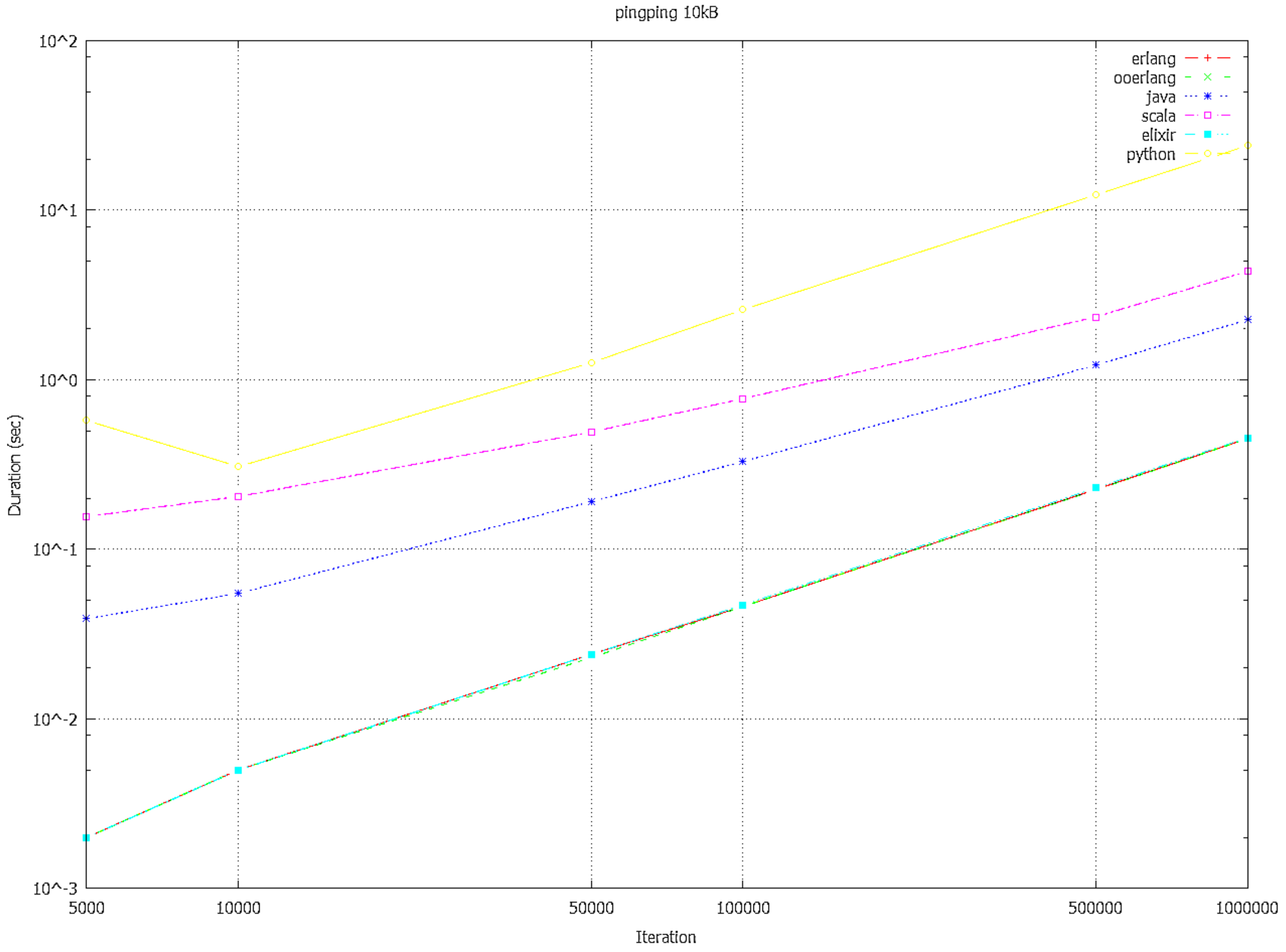
Experiment

- PingPing and PingPong
  - Iterations: 5K, 10K, 100K, 1M
  - Message Size: 5kB, 10kB, 50kB and 100kB

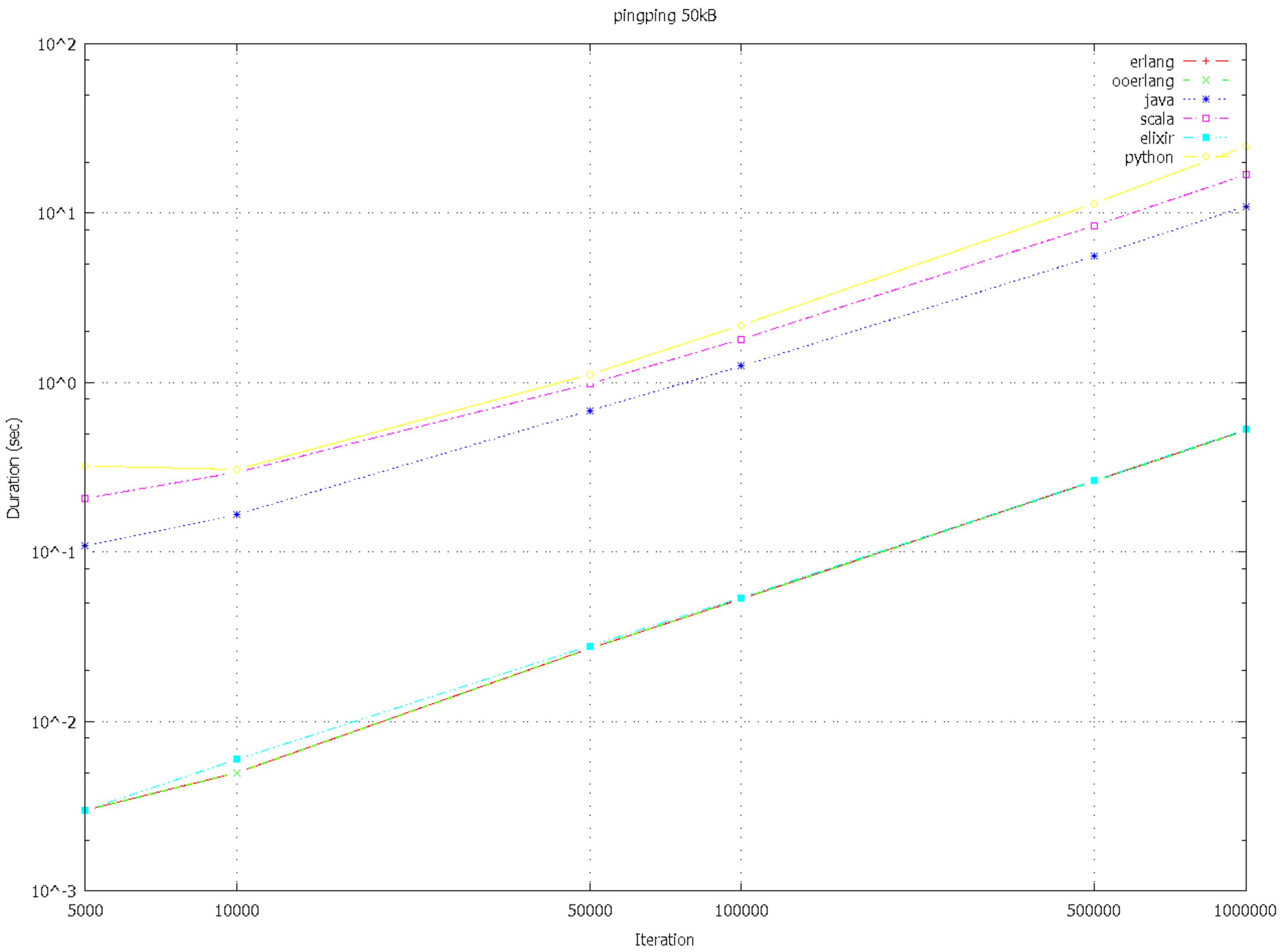
PingPing with messages 5kB



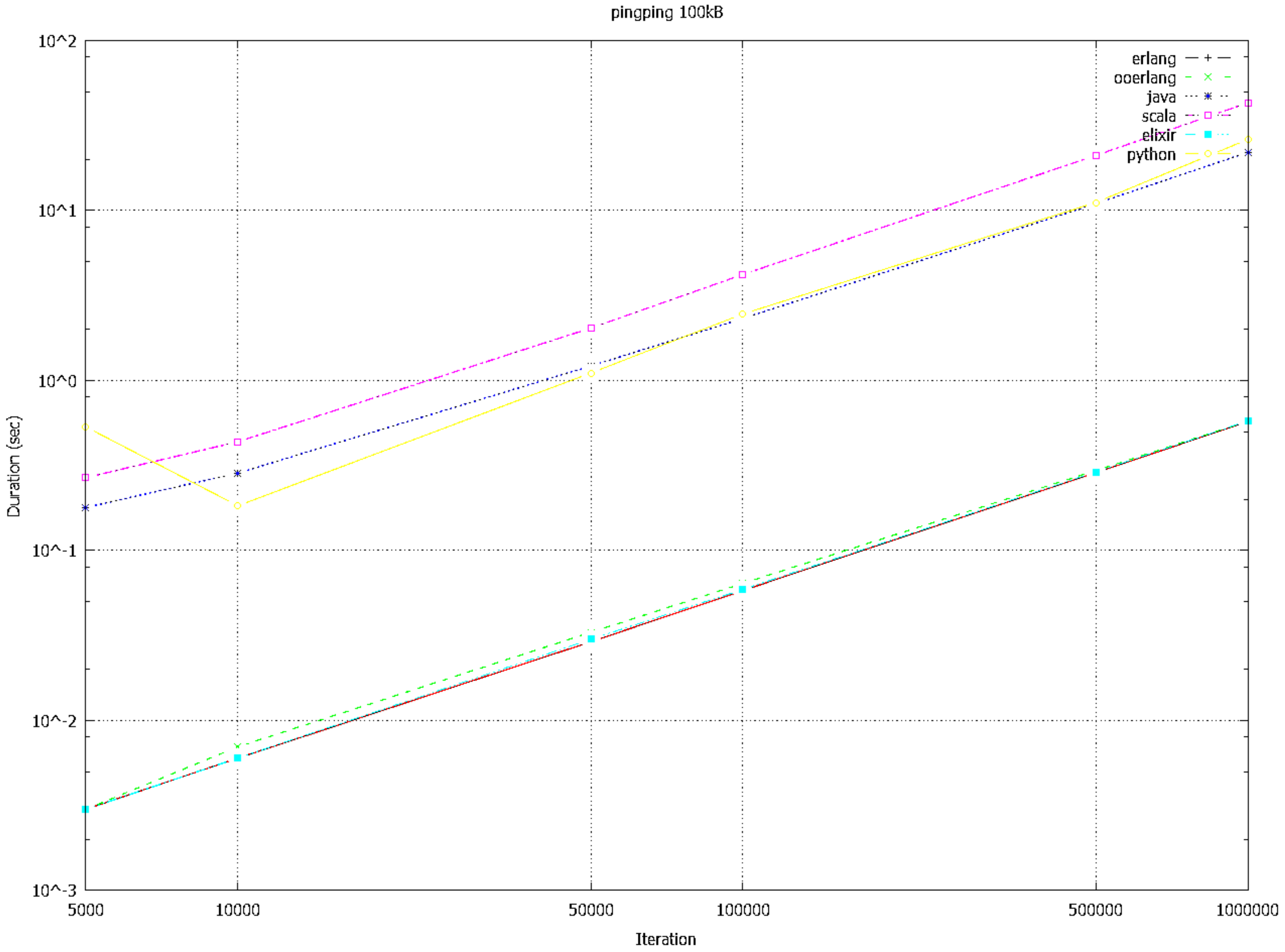
PingPing with messages 10kB



PingPing with messages 50kB



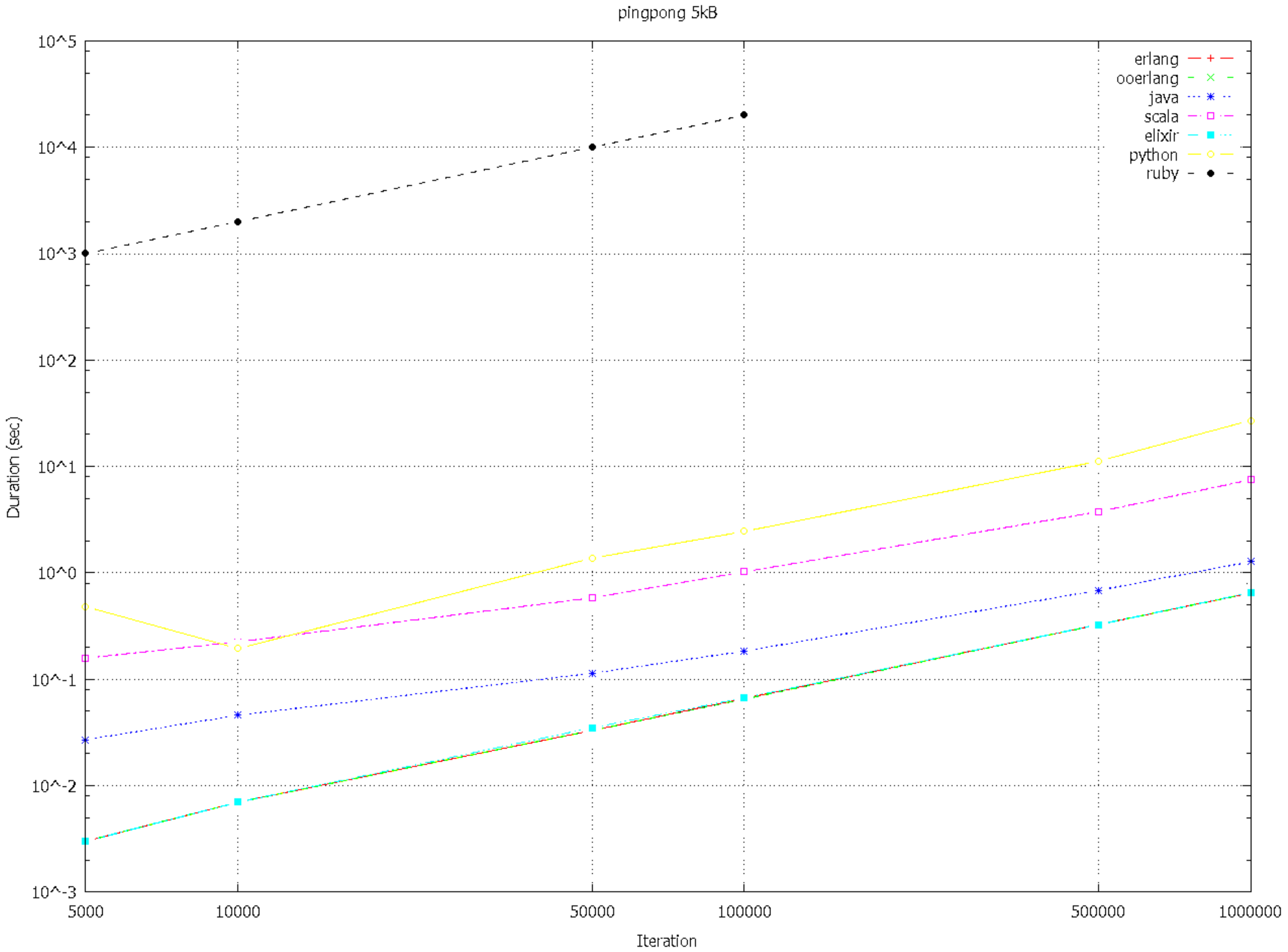
PingPing with messages 100kB



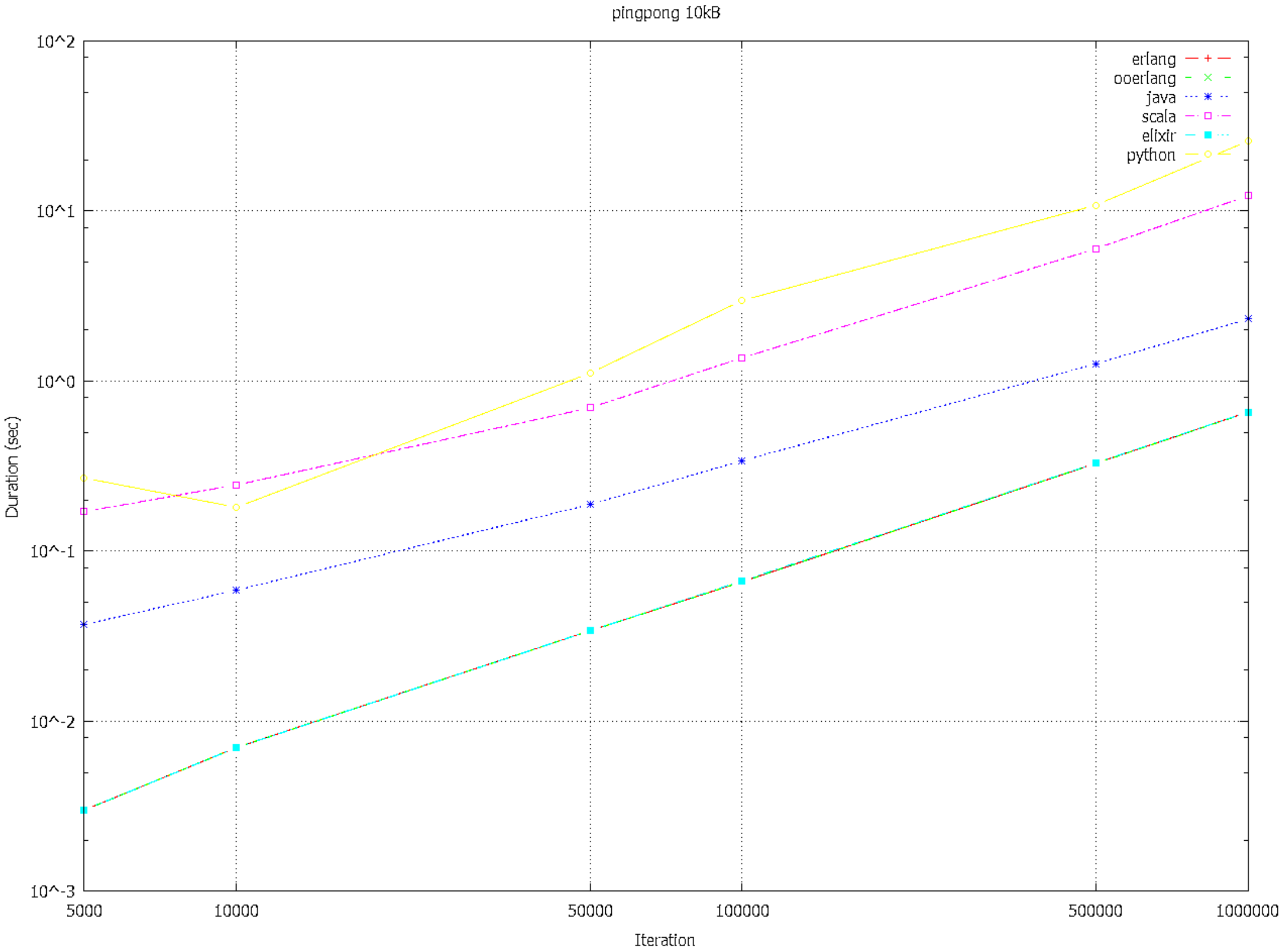
PingPong

```
def run(size, r) do

    data = generate_data(size)
    spawnStart = time_microseg()

    parent = self()
    p1 = spawn(fn -> pingping(data, parent, r) end)
    p2 = spawn(fn -> pingping(data, parent, r) end)

    spawnEnd = time_microseg()
    timeStart = time_microseg()
    send(p1, {:init, self, p2})
    finalize(p1)
    timeEnd = time_microseg()
    totalTime = timeEnd - timeStart
    spawnTime = spawnEnd - spawnStart
end
```

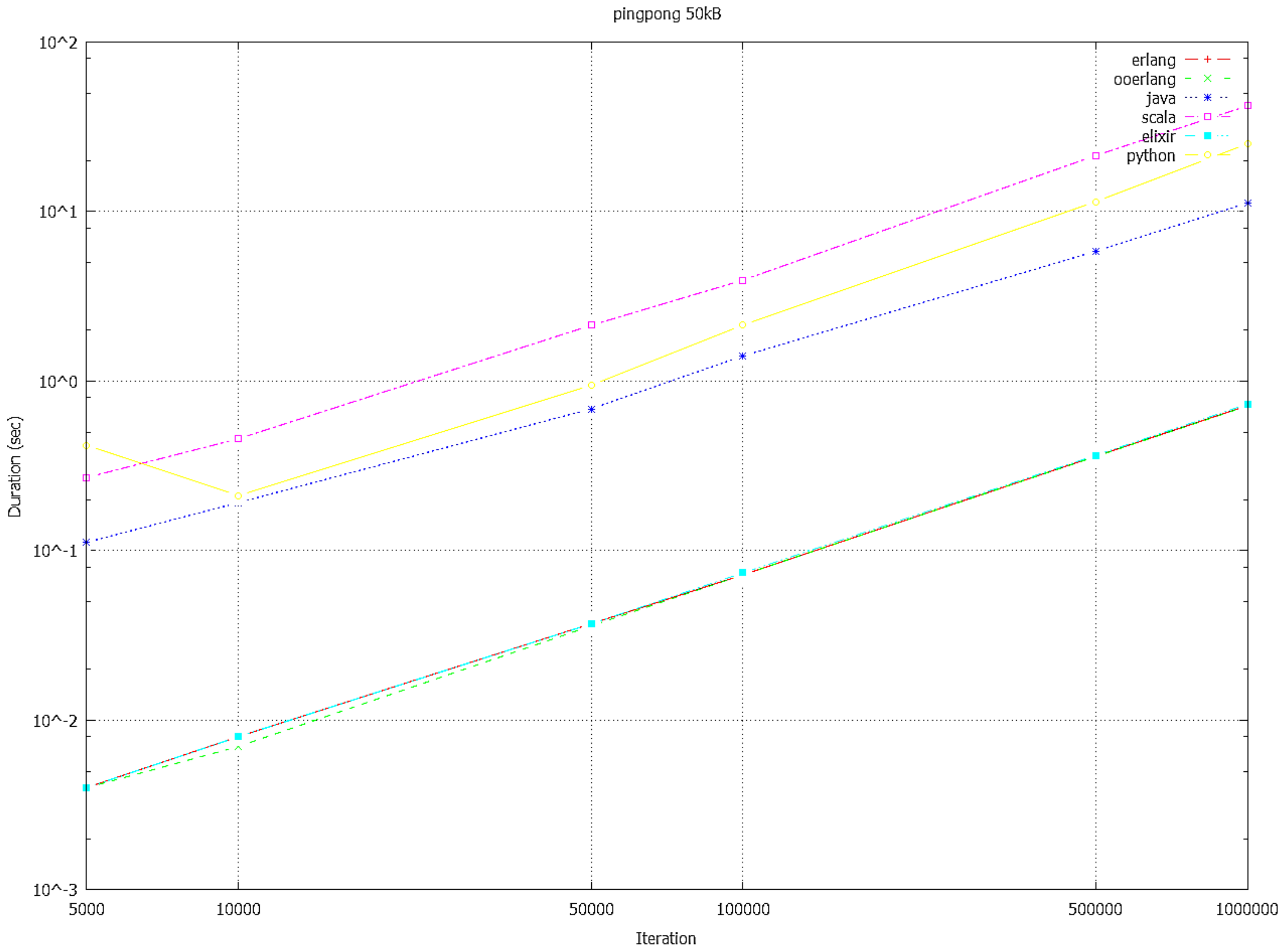
PingPong with messages 5kB



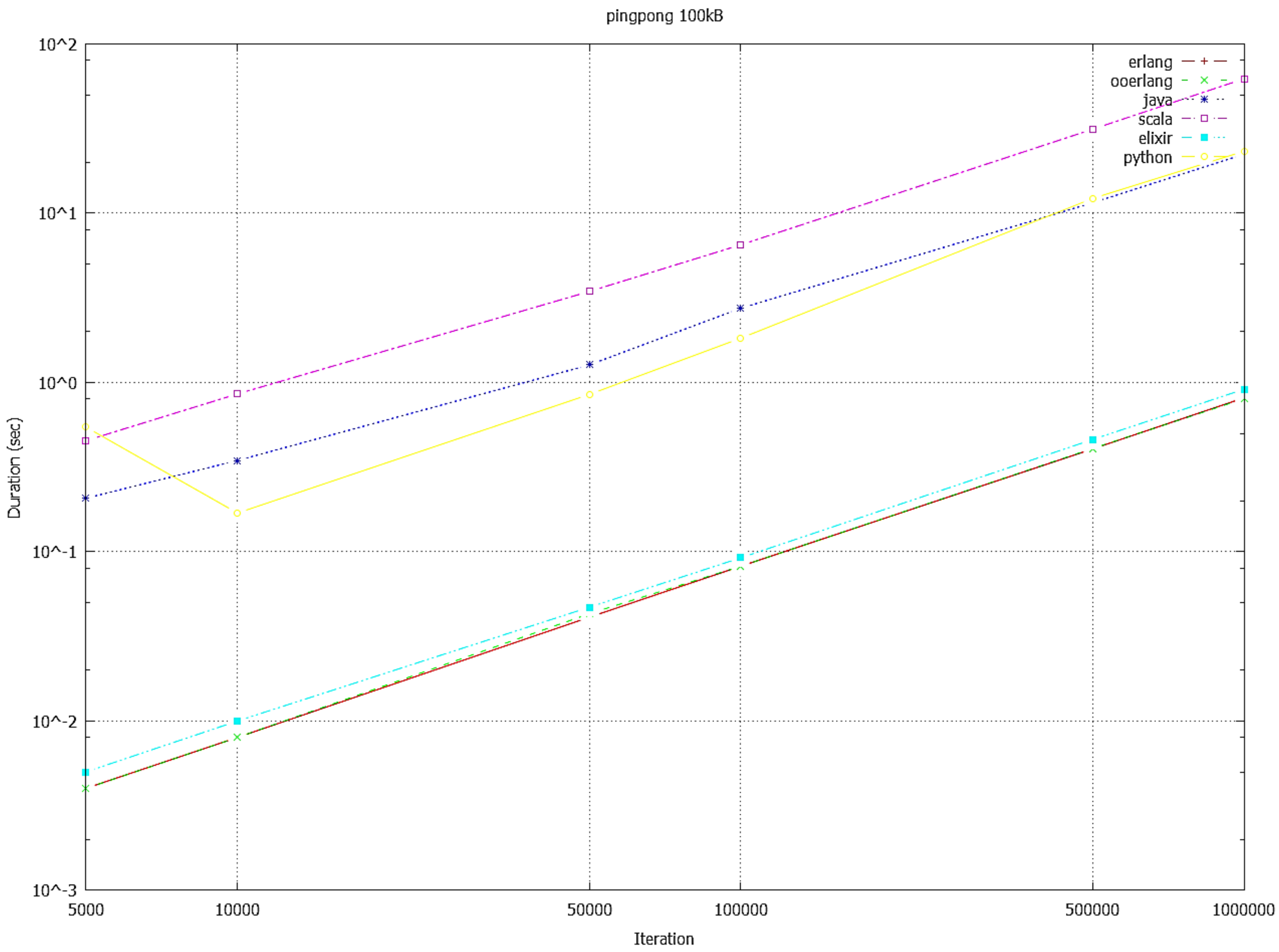
PingPong with messages 10kB



PingPong with messages 50kB



PingPong with messages 100kB



SendRecv

```
def run(data_size, rep, qtd_procs) do

    data = generate_data(data_size)

    spawn_start = time_microseg()
    second = create_procs(qtd_procs)
    spawn_end = time_microseg()

    exec_start = time_microseg()
    sender_ring_node(data, rep, second)
    exec_end = time_microseg()

    total_time = exec_end - exec_start
    spawn_time = spawn_end - spawn_start
end
```

SendRecv

```elixir
def ring_node(right_peer) do
  receive do
    data ->
      right_peer |> send(data)
      ring_node(right_peer)
  end
end

def create_procs(qtd_procs) do
  List.foldl(
    :lists.seq(qtd_procs, 2, -1),
    self,
    fn(_id, right_peer) -> spawn(__MODULE__, :ring_node, [right_peer]) end
  )
end
```

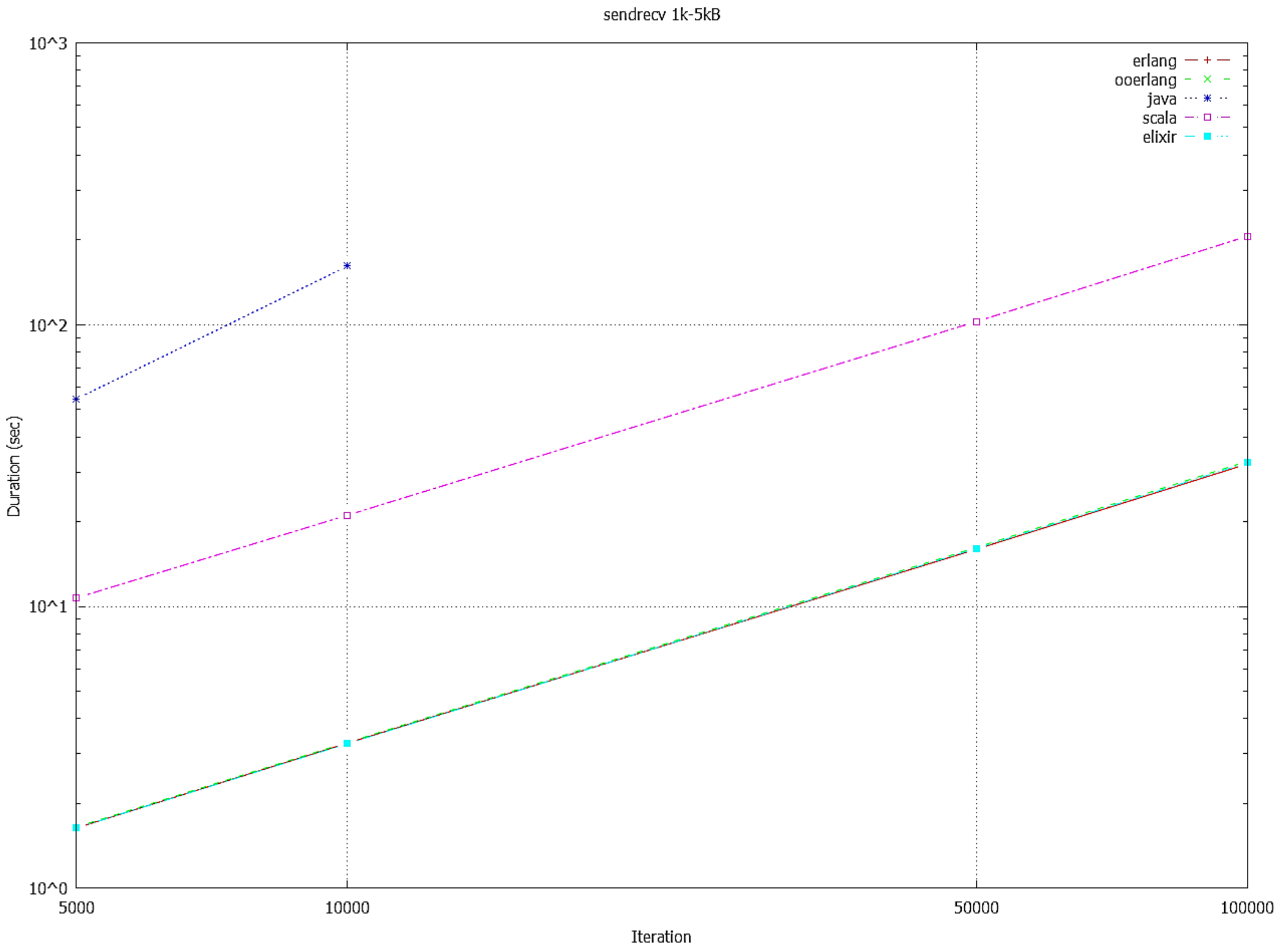
SendRecv

```
def sender_ring_node(_, 0, _) do
  :ok
end

def sender_ring_node(data, rep, second) do
  second |> send(data)
  receive do
    ^data ->
      sender_ring_node(data, rep - 1, second)
  end
end
```

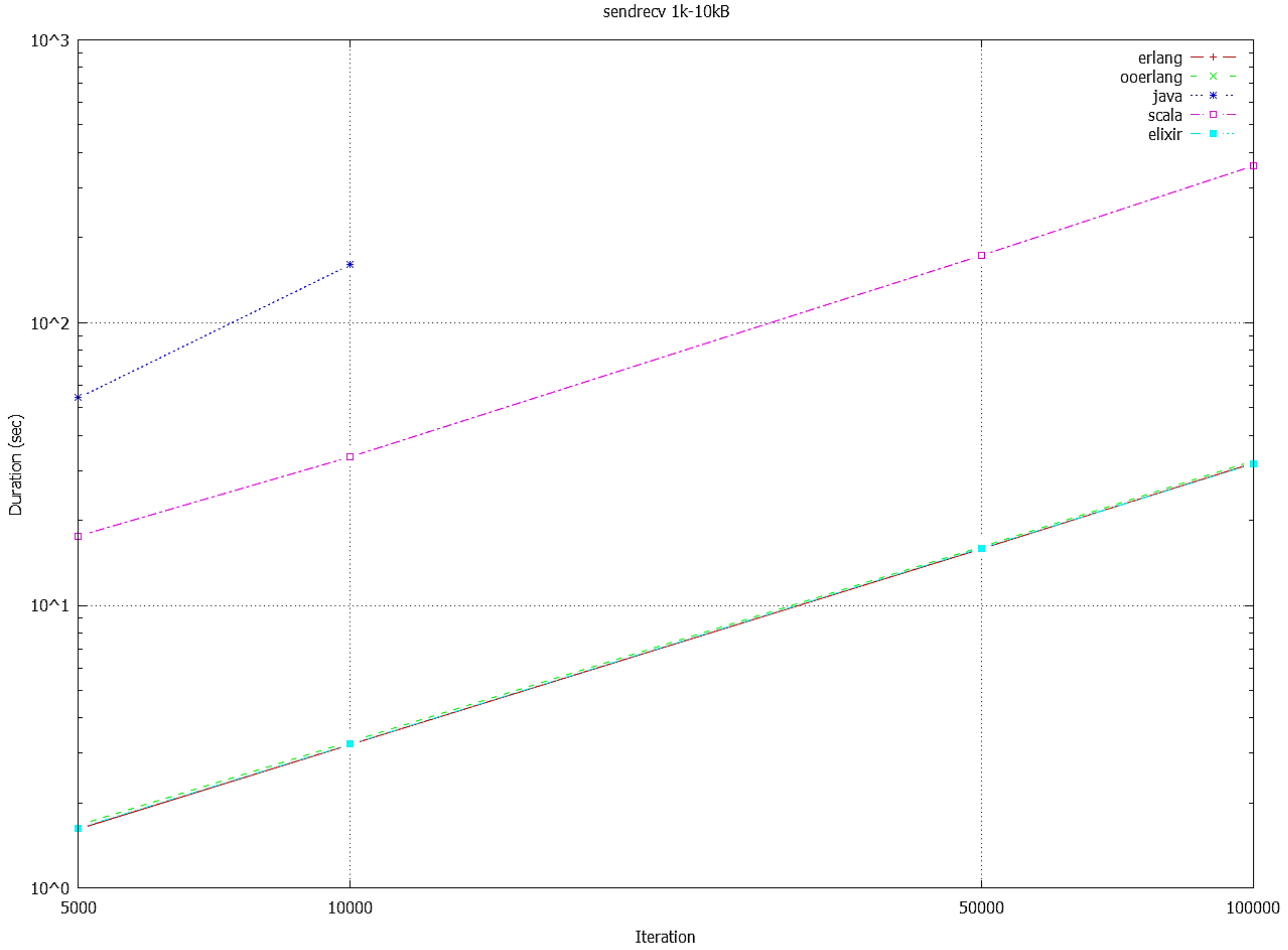
Experiment

- SendRecv
  - Number of Process: 1K, 10K and 50K
  - Iterations: 5K, 10K, 100K, 500K
  - Message Size: 5kB, 10kB, 50kB

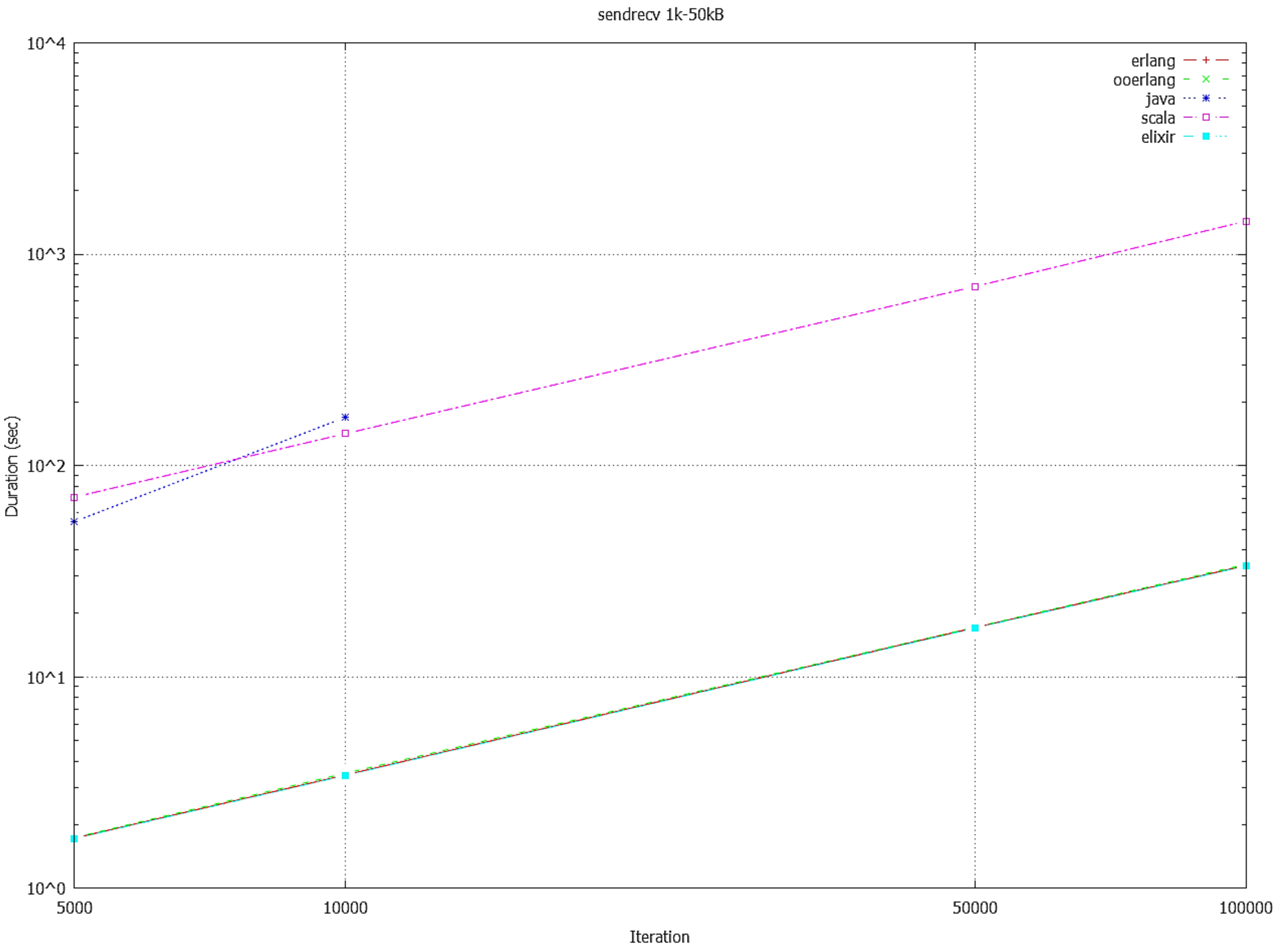
SendRec with 1,000 process and messages 5kB



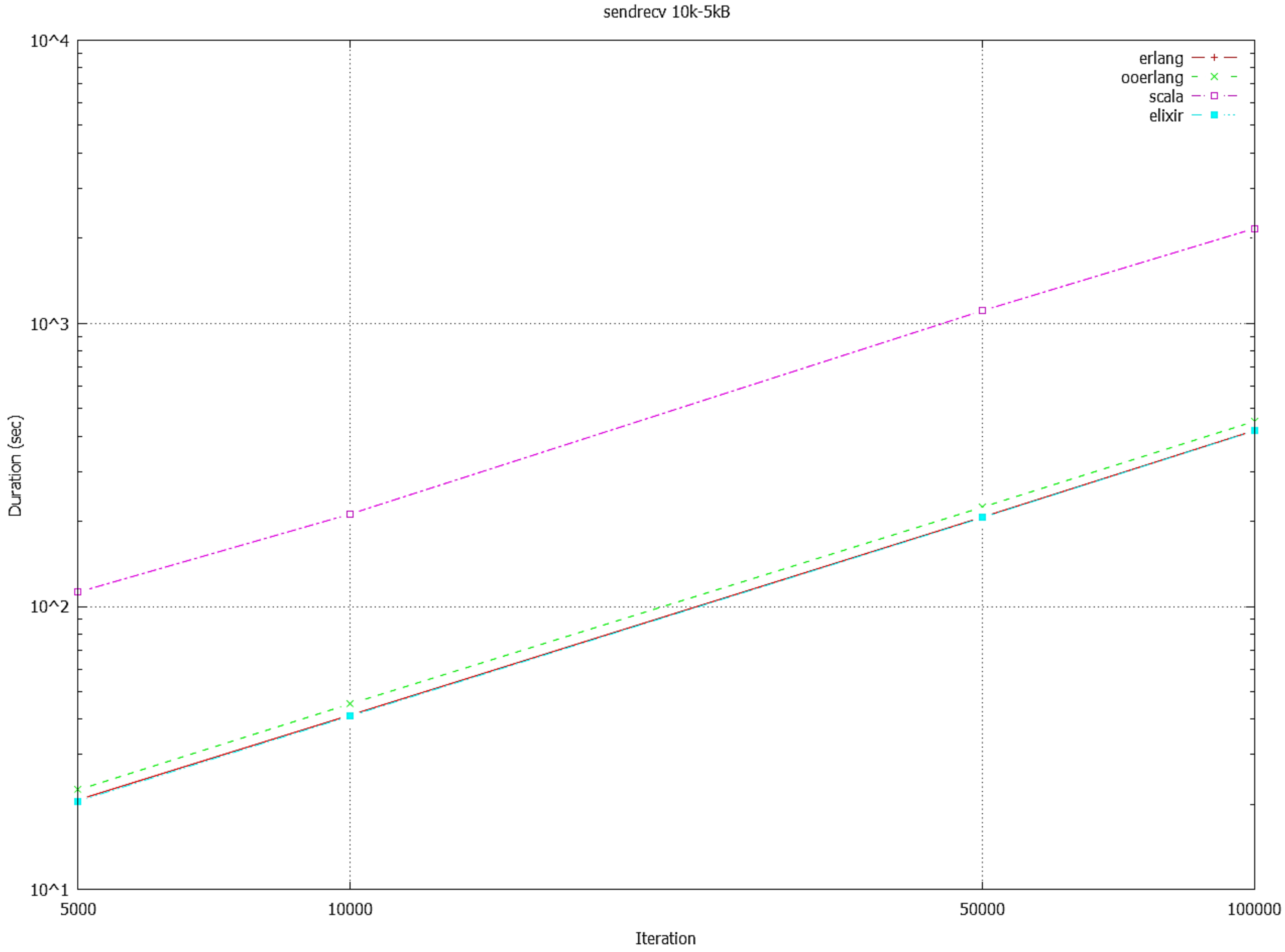
SendRec with 1,000 process and messages 10kB



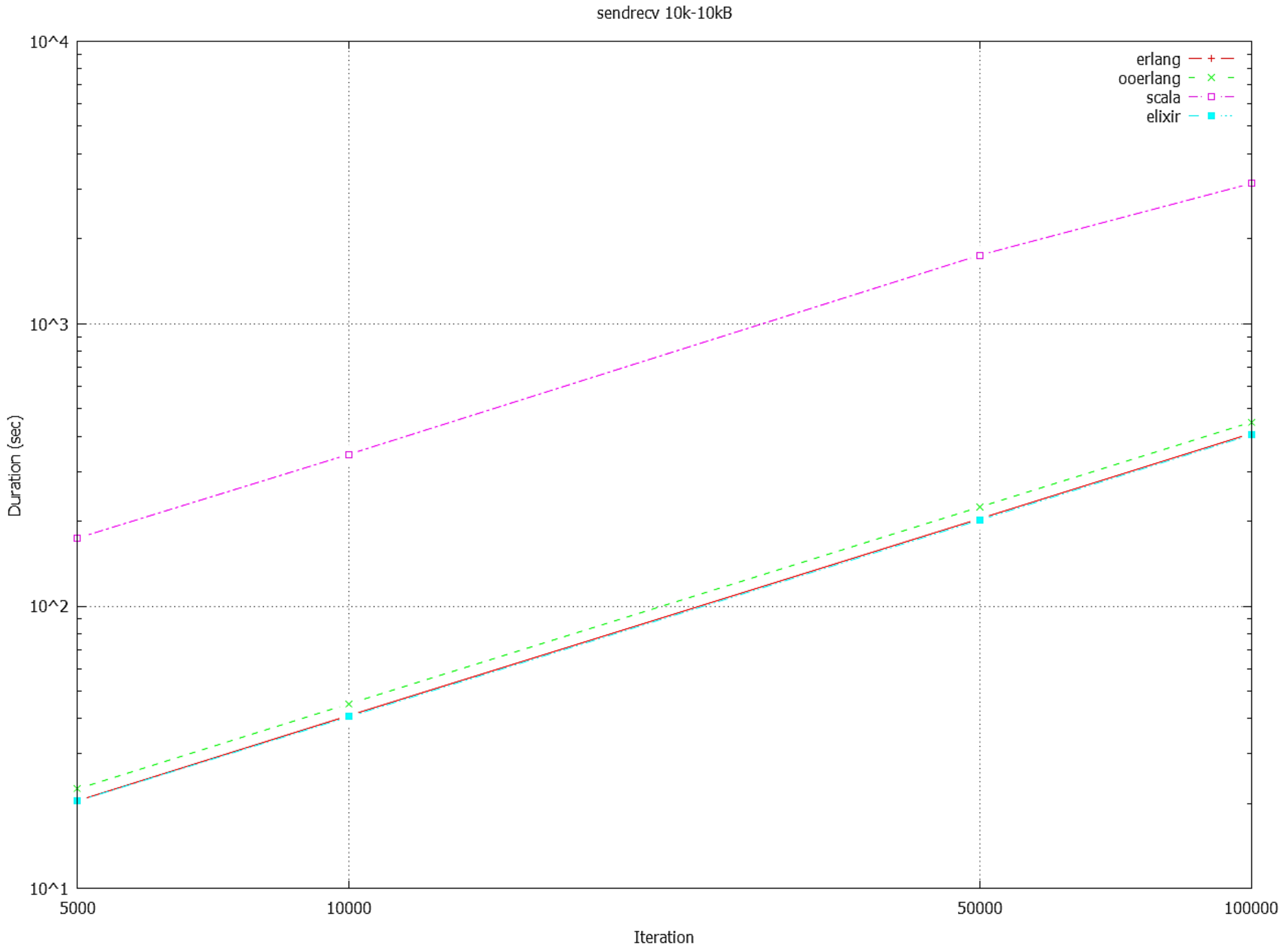
SendRec with 1,000 process and messages 50kB



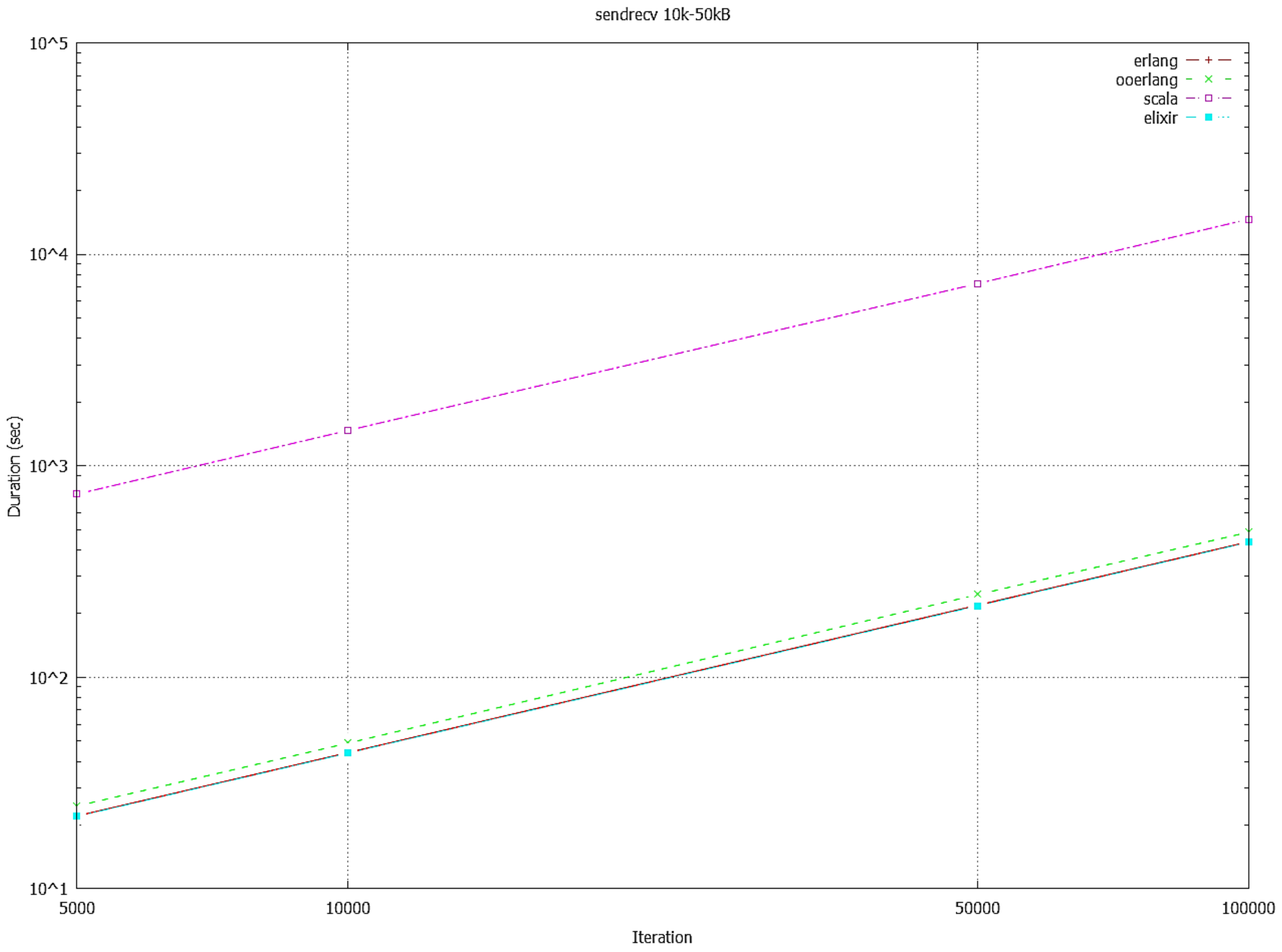
SendRec with 10,000 process and messages 5kB



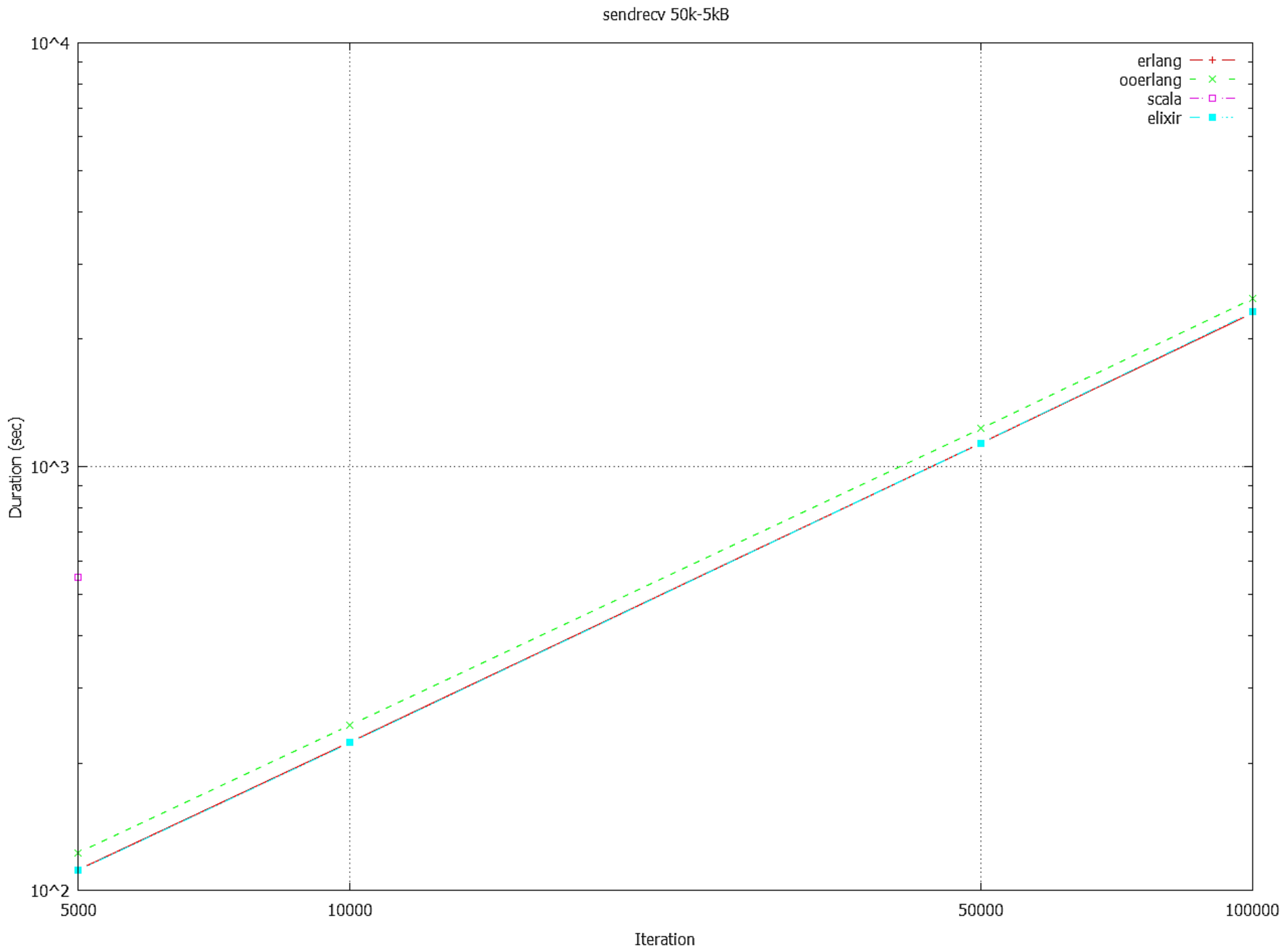
SendRec with 10,000 process and messages 10kB



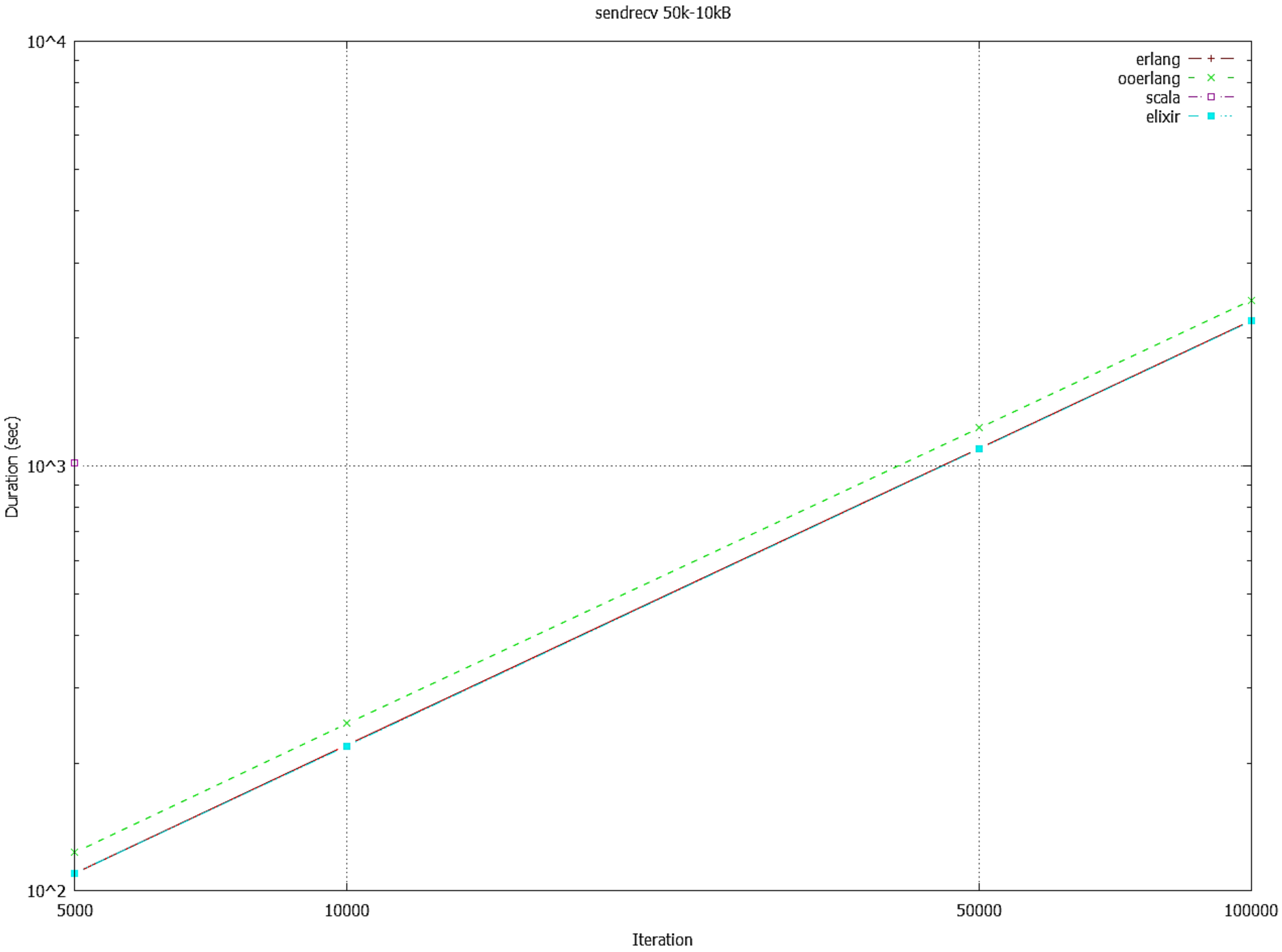
SendRec with 10,000 process and messages 50kB



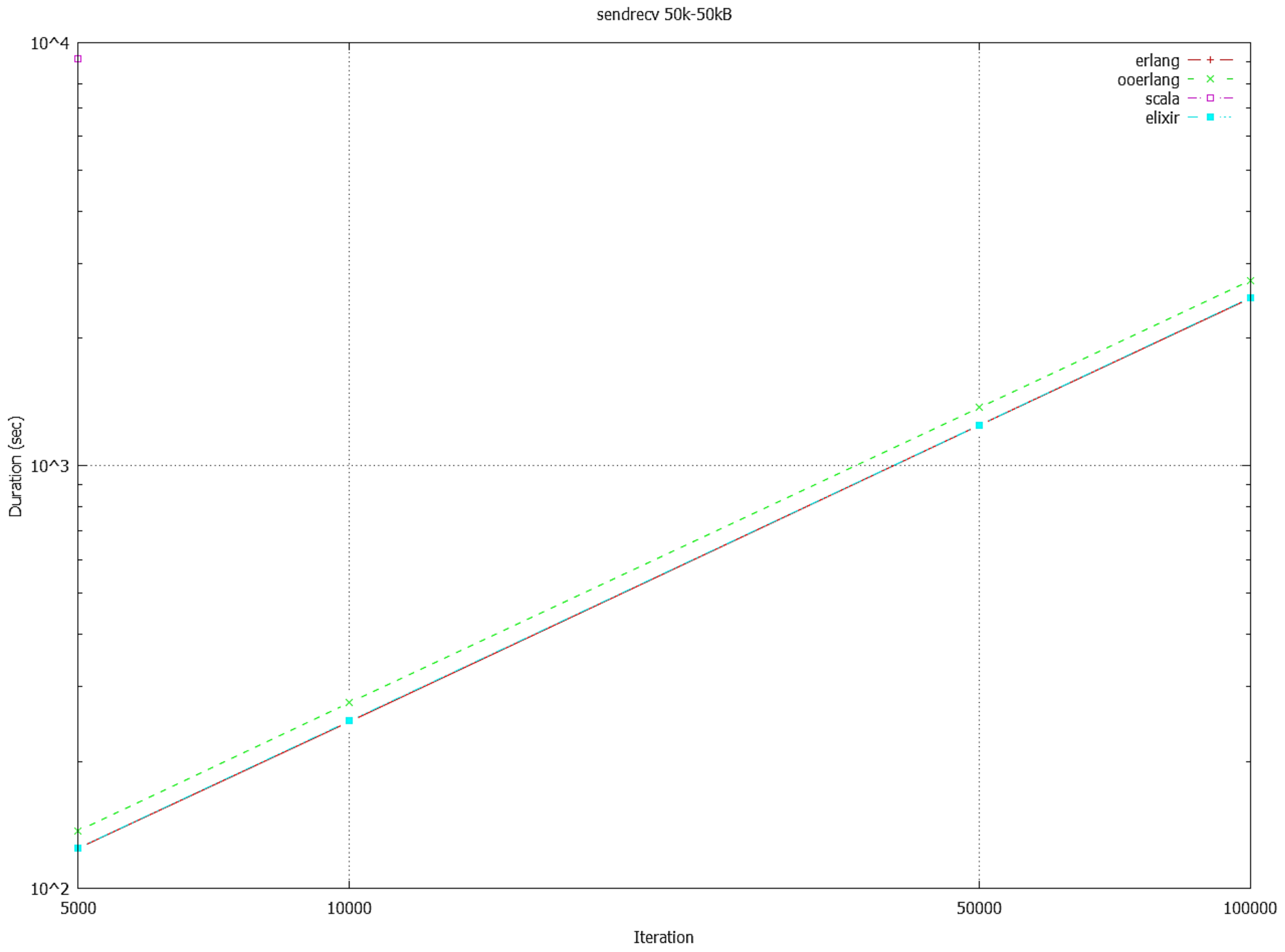
SendRec with 50,000 process and messages 5kB



SendRec with 50,000 process and messages 10kB



SendRec with 50,000 process and messages 50kB



The oscars of Most Communicative Language goes to …

- Erlang
- Elixir
- ooErlang

The oscars of Most Communicative Language goes to ...

- Clear superiority of Erlang over others languages

The oscars of Most Communicative Language goes to …

- Clear superiority of Erlang over others languages
- Elixir and ooErlang inherits Erlang's performance

The oscars of Most Communicative Language goes to ...

- Clear superiority of Erlang over others languages
- Elixir and ooErlang inherits Erlang’s performance
- Erlang, Elixir and ooErlang: limited only by the host machine and use all available resources

Results

- Scala, Java, Python and Ruby: Unable to run the test to completion
- Stopped working after 10 thounsands processes
- Do not use all available resources of the machine

Results

- Scala, Java, Python and Ruby: Unable to run the test to completion
- Stopped working after 10 thounsands processes
- Do not use all available resources of the machine

but ..

- Intel MPI Benchmark measures the communication and not the granularity of the languages

Future

- The Computer language Benchmarks Game http://benchmarksgame.alioth.debian.org/
  - n-body
  - fannkuch-redux
  - meteor-contest
  - fasta
  - spectral-norm
  - reverse-complement
  - mandelbrot
  - k-nucleotide
  - regex-dna
  - pidigits
  - chameneos-redux
  - thread-ring
  - binary-trees

Contributors

- Filipe Rafael Gomes Varjão

varjaofilipe@gmail.com / frgv@cin.ufpe.br

Filipergv

@filipevarjao

- Rafael Dueire Lins
- Jucimar Maia da Silva Jr.
- Emiliano Carlos de Moraes Firmino
- Francisco Heron de Carvalho Jr.
- Benjamin Tan Wei Hao (Elixir code review)
- José Valim (Elixir code review)