

Message-passing concurrency in Python

Sarah Mount – @snim2

Europython 2014

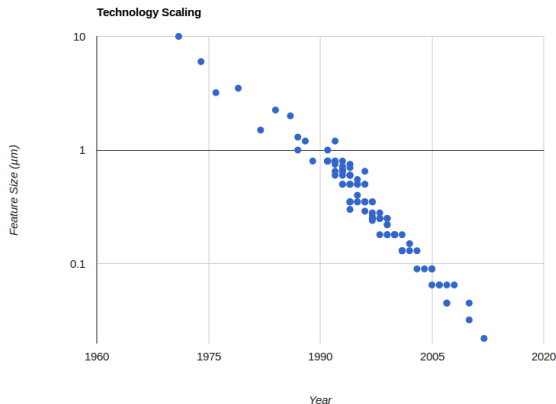
- 1 Why multicore?
- 2 Message passing: an idea whose time has come (back)
- 3 Message passing: all the cool kids are doing it
- 4 Design choices in message passing languages and libraries
- 5 Message passing in Python too!

Section 1

Why multicore?

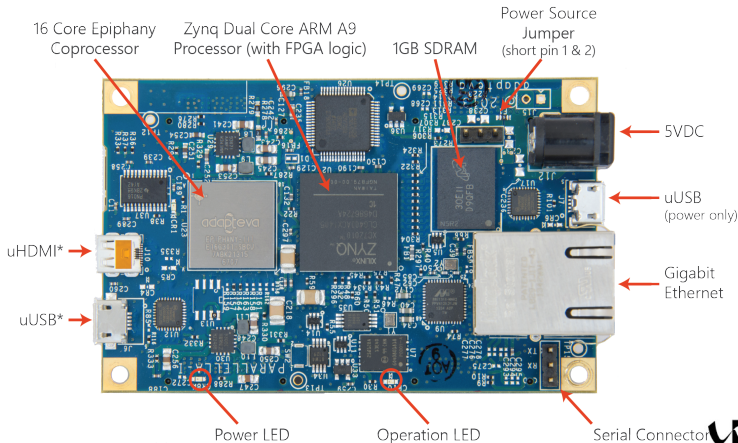
Why multicore? (1/2)

Feature size (nm) vs time (year). CPU sizes are shrinking! Get the raw data here: <http://cpudb.stanford.edu/>



Why multicore? (2/2)

New platforms, such as this board from Adapteva Inc. which reportedly runs at 50GFLOPS/Watt. Photo ©Adapteva Inc.

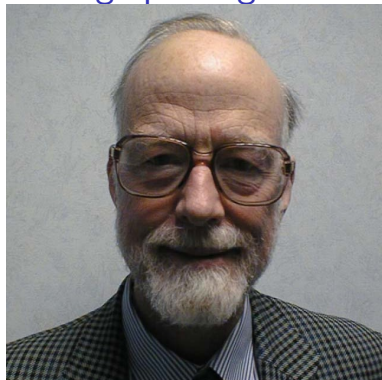


* Not on P1600-xxx models

Section 2

Message passing: an idea whose time has come (back)

Message passing concurrency



Tony Hoare

CSP^a (Communicating Sequential Processes) was invented by Tony Hoare in the late 70s / early 80s as a response to the difficulty of reasoning about concurrent and parallel code. It was implemented as the OCCAM programming language for the INMOS Transputer.

^aother process algebras are available

Message passing concurrency

CSP¹ take the view that computation consists of **communicating** processes which are individually made up of **sequential** instructions. The communicating processes all run "at the same time" and **share no data**. Because they do not share data they have to cooperate by **synchronising** on **events**. A common form of synchronisation (but not the only one) is to pass data between processes via **channels**.

THINK UNIX processes communicating via pipes.

BEWARE CSP is an abstract ideas which use overloaded jargon, a *process* or an *event* need not be reified as an OS process or GUI event. . .

¹other process algebras are available

Benefits of message passing

Threads and locking are tough! Deadlocks, starvation, race hazards are tough. Locks do not compose easily, errors in locking cannot easily be recovered from, finding the right level of granularity is hard.

CSP does not exhibit race hazards, there are no locks, the hairy details are all hidden away in either a language runtime or a library. WIN!

Section 3

Message passing: all the cool kids are doing it

```
$ cat mydata.csv | sort | uniq | wc -l
```

```
func main() {  
    mychannel := make(chan string)  
    go func() {  
        mychannel <- "Hello world!"  
    }()  
    fmt.Println(<-mychannel)  
}
```

Rust

```
fn main() {  
    let (chan, port) = channel();  
  
    spawn(proc() {  
        chan.send("Hello world!");  
    });  
  
    println!("{:s}", port.recv());  
}
```

Scala

```
object Example {  
  class Example extends Actor {  
    def act = {  
      receive {  
        case msg => println(msg)  
      }  
    }  
  }  
  
  def main(args:Array[String]) {  
    val a = new Example  
    a.start  
    a ! "Hello world!"  
  }  
}
```

```
@process
def client(chan):
    print chan.read()


@process
def server(chan):
    chan.write('Hello world!')

if __name__ == '__main__':
    ch = Channel()
    Par(client(ch), server(ch)).start()
```

²<http://github.com/snim2/python-csp>

naulang³: building on the RPython toolchain

```
let f = fn(mychannel) {  
    mychannel <- "Hello world!"  
}  
  
let mychannel = chan()  
async f(mychannel)  
print <: mychannel
```

³Samuel Giles (2014) Is it possible to build a performant, dynamic language that supports concurrent models of computation? University of Wolverhampton BSc thesis 

naulang⁴: building on the RPython toolchain

Table : Results from a tokenring benchmark @SamuelGiles_

	Min (μ s)	Max (μ s)	Mean (μ s)	s.d.
Go 1.1.2	99167	132488	100656	1590
Occam- π	68847	73355	69723	603
naulang	443316	486716	447894	6507
naulang JIT	317510	589618	322304	10613
naulang (async)	351677	490241	354325	6401
naulang JIT (async)	42993	49496	44119	295

<http://github.com/samgiles/naulang>

⁴Samuel Giles (2014) Is it possible to build a performant, dynamic language that supports concurrent models of computation? University of Wolverhampton BSc thesis

Aside: Did those benchmarks matter?

That was a comparison between naulang and two compiled languages. The benchmark gives us some reason to suppose that a dynamic language (built on the RPython chain) can perform reasonably well compared to static languages.

Does this matter? We don't want message passing to become a bottleneck in optimising code...

Section 4

Design choices in message passing languages and libraries

Synchronous vs. asynchronous channels

Synchronous channels block on read / write - examples include OCCAM, JCSP, python-csp, UNIX pipes. **Asynchronous** channels are non-blocking - examples include Scala, Rust. Some languages give you both - examples include Go.

Synchronous channels are easier to reason about (formally) and many people believe are easier to use.

Asynchronous channels are sometimes thought to be faster. Like most claims about speed, this isn't always true.

Uni or bidirectional channels? Point to point?

The JCSP⁵ project (for Java) separates out different sorts of channels types, e.g.:

- AnyToAnyChannel
- AnyToOneChannel
- OneToAnyChannel
- OneToOneChannel

This isn't very Pythonic, but it can catch errors in setting up a network of channels in a message passing system.

⁵<http://www.cs.kent.ac.uk/projects/ofa/jcsp/>

Mobile or immobile channels

A **mobile** channels can be sent over another channel. This means that the network of channels can change its topology dynamically at runtime. Why might we want to do this?

- our process are actually running over a network of machines or on a manycore processor. We want to migrate some processes to perform load balancing or due to some failure condition within a network.
- some processes have finished their work and any channels associated with them need to be either **poisoned** (destroyed) or re-routed elsewhere.

Reifying sequential processes

In **CSP** and **actor model** languages the paradigm is of a number of *sequential processes* which run in parallel and communicate via channels (think: UNIX processes pipes). It is up to the language designer how to reify those sequential processes. Common choices are:

- 1 CSP process is 1 coroutine very fast message passing (OCCAM- π reports a small number of ns), cannot take advantage of multicore.
- 1 CSP process is 1 OS thread slower message passing; can take advantage of multicore
- 1 CSP process is 1 OS process very slow message passing; can migrate CSP processes across a network to other machines
- 1 CSP processes is 1 coroutine many coroutines are multiplexed onto an OS thread
- 1 CSP processes is 1 OS thread many OS threads are multiplexed onto an OS process

etc.

Section 5

Message passing in Python too!

So many EP14 talks!

Several EP14 talks involve coroutines (lightweight threads) and / or channels:

Daniel Pope Wednesday 11:00

Christian Tismer, Anselm Kruis Wednesday 11:45

Richard Wall Friday 11:00

Benoit Chesneau Friday 12:00

Dougal Matthew Friday 12:00

Message-passing has already entered the Python ecosystem and there are constructs such as pipes and queues in the multiprocessing library. Could we do better by adopting some ideas from other languages?

The (near) future

Python for the Parallella will be coming out this Summer, thanks to funding from the RAE. If you are interested in this, please keep an eye on the **futurecore**⁶ GitHub organization.

python-csp is moving back into regular development. Will it ever be fast?

⁶<http://github.com/futurecore/>

Thank you.