

Scaling PyCSP

Rune Møllegaard Friborg, John Markus Bjørndalen and Brian Vinter

CPA 2013, Edinburgh
August 25, 2013



AARHUS UNIVERSITY



UNIVERSITY OF
COPENHAGEN



Python for eScience Applications

- Flexible
- Can interface with most programming languages
- Many scientists already know Python
- Faster development cycle
- Compute-intensive parts written in compilable languages are easily integrated
- Forces programmers to write readable code

CSP for eScience!

- Synchronized constructs for running a set of processes
 - In parallel
 - In sequence
- Synchronized communication through message passing
 - One-way channels
- Complete Process Isolation
 - No shared data-structures
 - No side-effects from processes
 - Compositional structure
 - Reuse of processes
- The data flow in scientific applications is often simple to model in CSP. 3

Introduction to PyCSP

- 2007 - PyCSP is presented. The synchronization model for channel communications is based on JCSP.
- 2009 - A PyCSP with a new synchronization model is presented. It is using the two-phase locking protocol to allow any-to-any channels supporting both input and output guards.
- 2011-2012 - A distributed version of the synchronization model is presented and later implemented in PyCSP

We want to run anywhere!

- The user of PyCSP does not need to know anything about the location of the hardware any process might run on
- All channel ends are mobile

Basic PyCSP Features

Single Any-to-Any Channel

```
A = Channel("A")
```

Buffered

```
A = Channel("A", buffer=10)
```


Termination through Poisoning / Retiring

```
Cin = A.reader()
```

```
Cin.poison() # propagate pill right  
now!
```

```
Cin.retire() # propagate pill, when  
all readers on A have invoked retire
```

External Choice

Does not guarantee priority

AltSelect(InputGuard(cin), OutputGuard(cout, msg))

External Choice

Guarantees priority, by adding a wait for an acknowledgement

PriSelect (InputGuard(cin) , OutputGuard(cout, msg))

External Choice

```
# Uses PriSelect to perform a fair choice through  
reordering of guards, based on past selections
```

```
FairSelect(InputGuard(cin) , OutputGuard(cout, msg) )
```

Declaring Processes

```
# An OS thread
@process
def Increment(cin, cout):
    cout(cin() + 1)
```

Declaring Processes

```
# An OS process
@multiprocess
def Increment(cin, cout):
    cout(cin() + 1)
```

Starting Processes

```
# Blocking PAR - Natural number generator
Parallel (
  Prefix(C.reader(), A.writer(), 1),
  Increment(A.reader(), B.writer(),
  Delta(B.reader(), C.writer(), D.writer()))
)

Spawn(processes...)

Sequence(processes...)
```

Compositional

```
@process
def Counter(cout) :
  Parallel(
    Prefix(C.reader(), A.writer(), 1),
    Increment(A.reader(), B.writer(),
    Delta(B.reader(), C.writer(), cout)
  )
)
```


Connecting Channels Host A

```
# Hosting channel A
A = Channel("A")

# Get address
print(A.address)
('192.168.1.16', 63550)
```

Connecting Channels Host B

```
# Connect to channel A  
A = Channel("A", connect=('192.168.1.16', 63550))
```

@clusterprocess

Declaring Remote Process

```
# A cluster process
@clusterprocess
def Increment(cin, cout):
    cout(cin() + 1)
```

Declaring Remote Process

```
# A cluster process
@clusterprocess(
    cluster_nodefile = <file containing list of nodes>,
    cluster_pin      = <index for node in list>,
    cluster_hint     = <'blocked' or 'strided'>
)
def Increment(cin, cout):
    cout(cin() + 1)
```

Executing Remote Process

```
# Spawn single increment process
```

```
Spawn(Increment(A.reader(), B.writer()))
```

```
# or
```

```
# Spawn 5 increment processes and put them on 5  
different hosts if available
```

```
Spawn( 5 * Increment(A.reader(), B.writer()),  
       cluster_hint = 'strided')
```

Connecting Channels Implicitly

```
# Blocking PAR - Natural number generator
# One clusterprocess per host
Parallel(
  Prefix(C.reader(), A.writer(), 1),
  Increment(A.reader(), B.writer(),
Delta(B.reader(), C.writer(), D.writer()),
  cluster_hint = 'strided'
)
```

Connecting Channels Implicitly

```
@clusterprocess  
def P1(cin):  
    cin() # read value
```

```
@clusterprocess  
def P2(cout):  
    cout(42) # send value
```

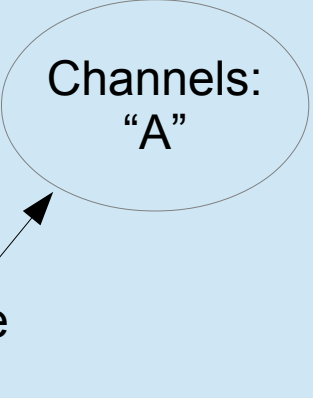
```
A = Channel()  
Parallel(P1(A.reader()),P2(A.writer()))
```


Connecting Channels Implicitly

```
@clusterprocess
def P1(cin):
    cin() # read value

@clusterprocess
def P2(cout):
    cout(42) # send value

A = Channel("A")
Parallel(P1(A.reader()),P2(A.writer()))
```



The diagram illustrates the implicit connection between two processes, P1 and P2, through a shared channel named 'A'. The channel is represented by an oval labeled 'Channels: "A"'. An arrow points from the 'A' argument in the 'Parallel' function call to this oval, indicating that both processes are connected to the same channel.

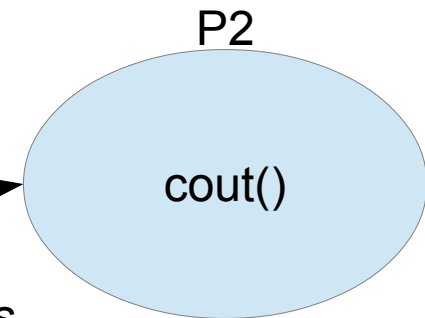
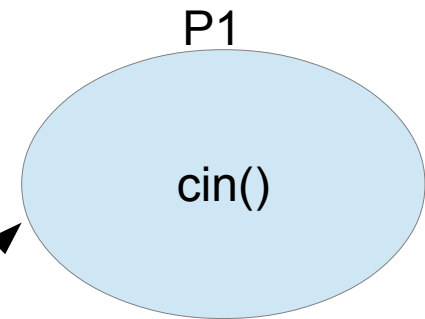
Connecting Channels Implicitly

```
@clusterprocess
def P1(cin):
    cin() # read value

@clusterprocess
def P2(cout):
    cout(42) # send value

A = Channel("A")
Parallel(P1(A.reader()),P2(A.writer()))
```

Channels:
"A"



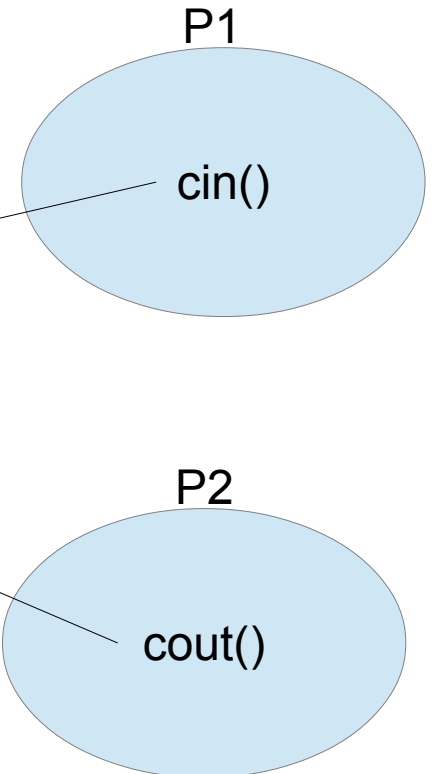
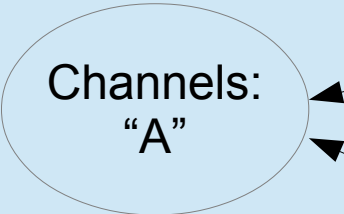
Starting processes on remote hosts using the SSH protocol. PyCSP channels are used to transfer any function parameters

Connecting Channels Implicitly

```
@clusterprocess
def P1(cin):
  cin() # read value

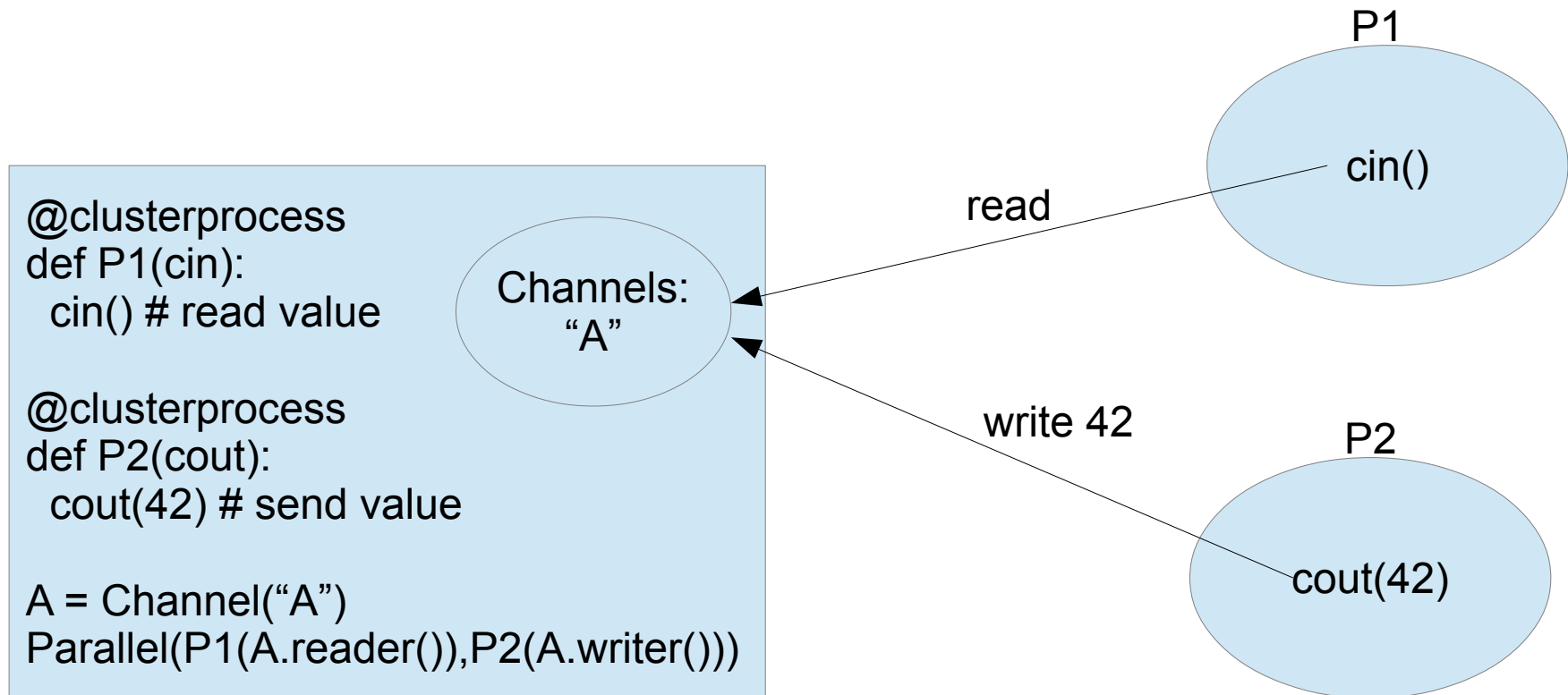
@clusterprocess
def P2(cout):
  cout(42) # send value

A = Channel("A")
Parallel(P1(A.reader()),P2(A.writer()))
```



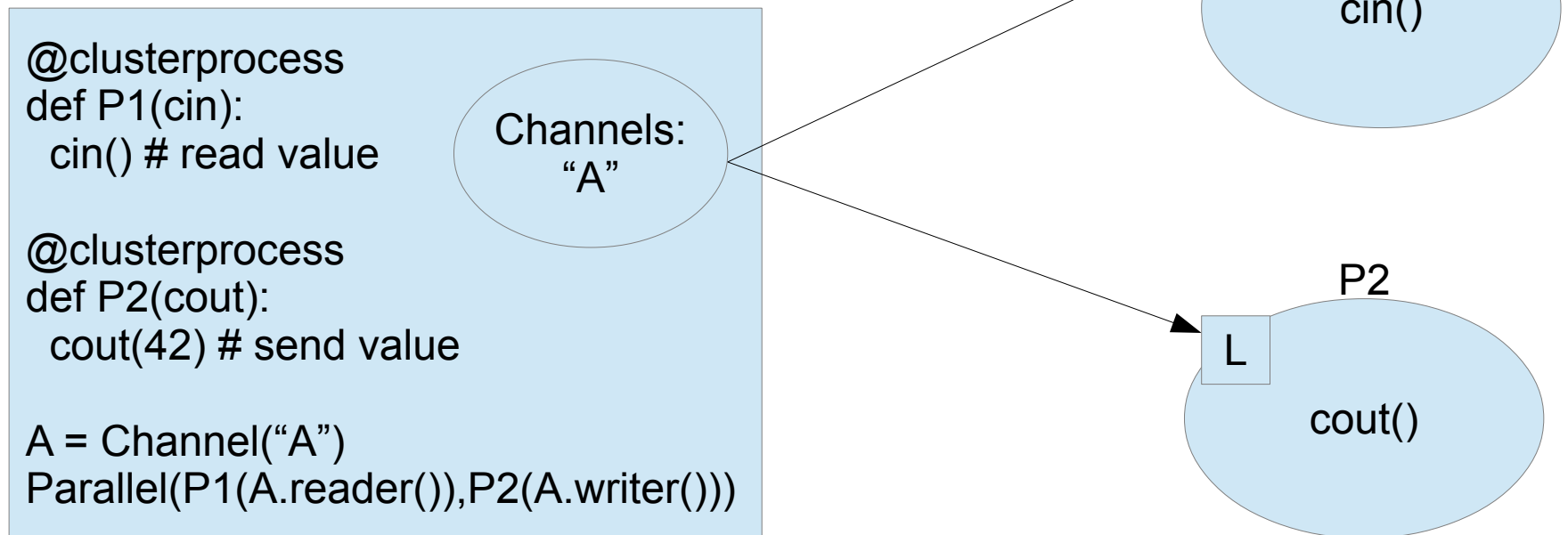
The channel ends cin and cout reconnect to the channel home and registers as they are both new channel ends.

Connecting Channels Implicitly



The channel ends cin and cout now posts a request for communication at the channel home

Connecting Channels Implicitly



The channel home then probes a read and write request for a potential match.

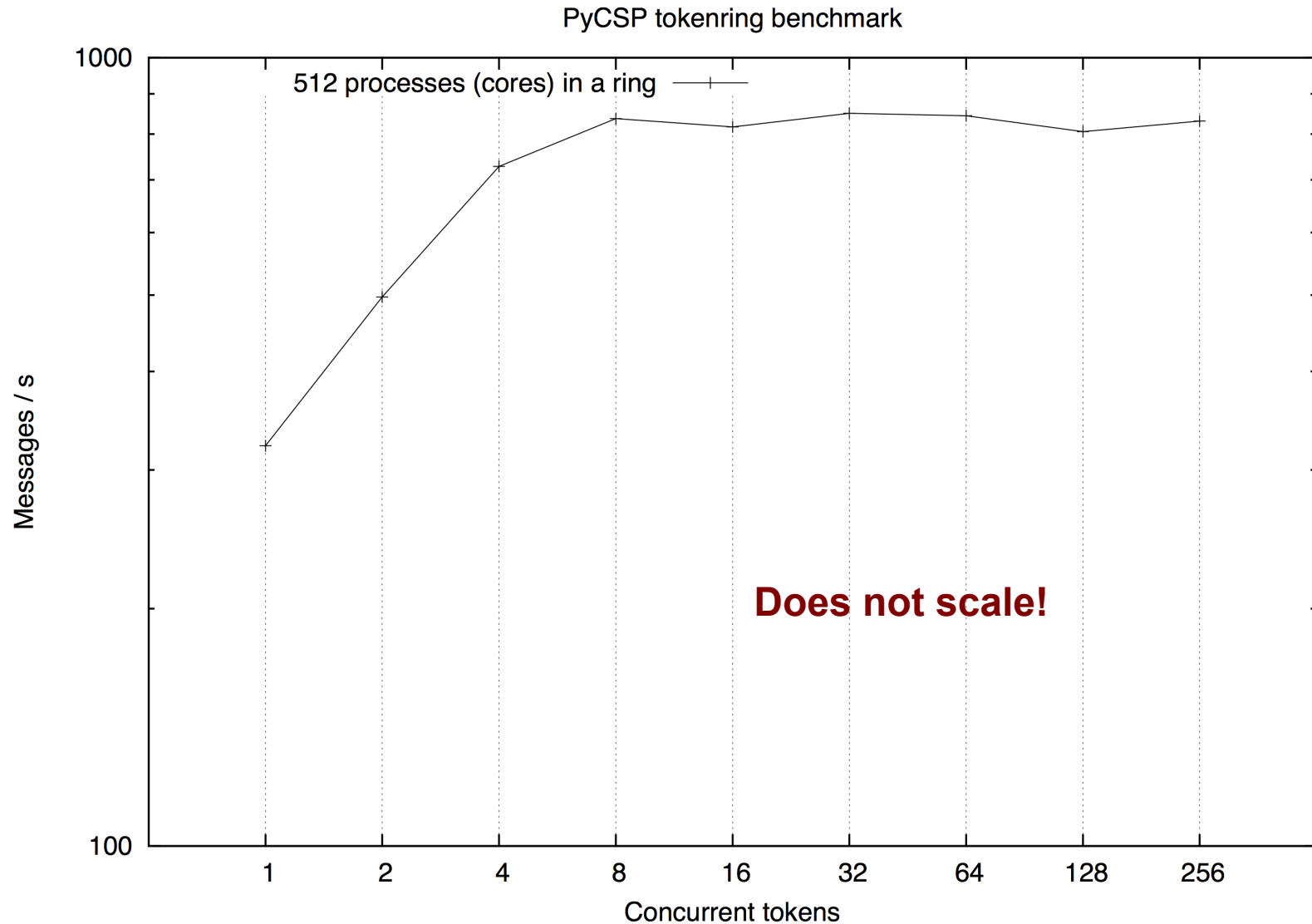
Acquires the process locks and if successful, transfers the messages and notifies the processes

512 processes (cores) in a ring

```
@clusterprocess
def elementP(this_read, next_write):

    while True:
        token = this_read()
        next_write(token + 1)
```

512 processes (cores) in a ring



Possible solutions

- Avoid a channel home completely
 - Requires a lot more messages for any-to-any channels. The location of all processes connected to a channel must always be known.
- Request the user to redistribute another set of channels, where each channel is hosted evenly on the available hosts
 - Difficult for the user
- Add mobility to a channel home, such that it may be moved during active use.
 - Simple for the user. Our choice.

Introducing Mobile Channel Homes in PyCSP

```
@clusterprocess
def elementP(this_read, next_write):

    this_read.become_home()

    while True:
        token = this_read()
        next_write(token + 1)
```

Introducing Mobile Channel Homes in PyCSP

- Based on a transition model presented in 2011.
 - When a channel is poisoned, all active requests (processes) at a channel are notified with a POISON signal
 - Similarly, when a channel home is moved, all active requests (processes) at a channel are notified with a MOVE signal
 - Processes then receive the new address of the channel together with the MOVE signal. The processes then withdraws the active request from the “old” channel home and reposts the request at the new channel home.

Introducing Mobile Channel Homes in PyCSP

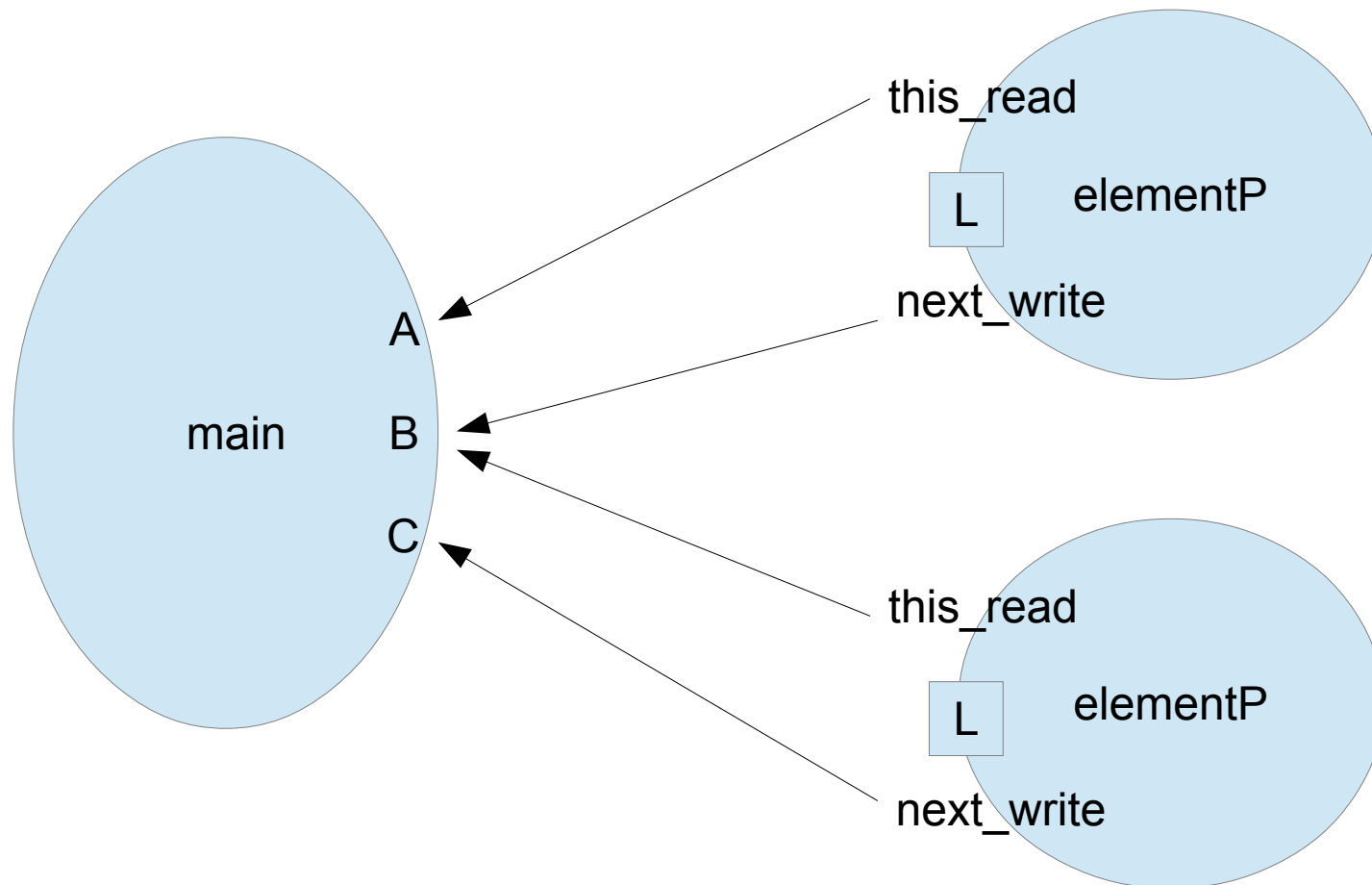
- Based on a transition model presented in 2011

```
@clusterprocess
def elementP(this_read, next_write):

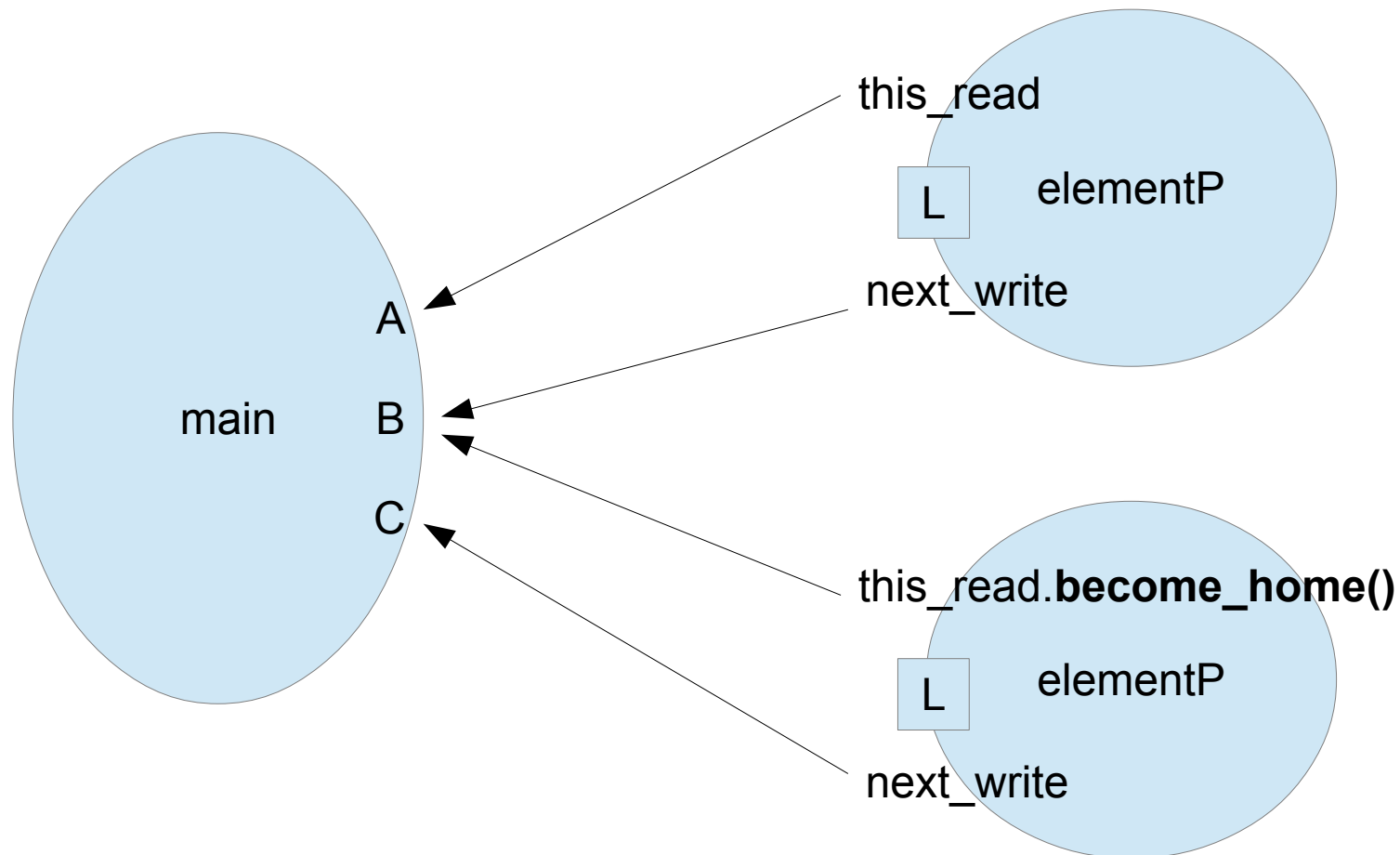
    this_read.become_home()

    while True:
        token = this_read()
        next_write(token + 1)
```

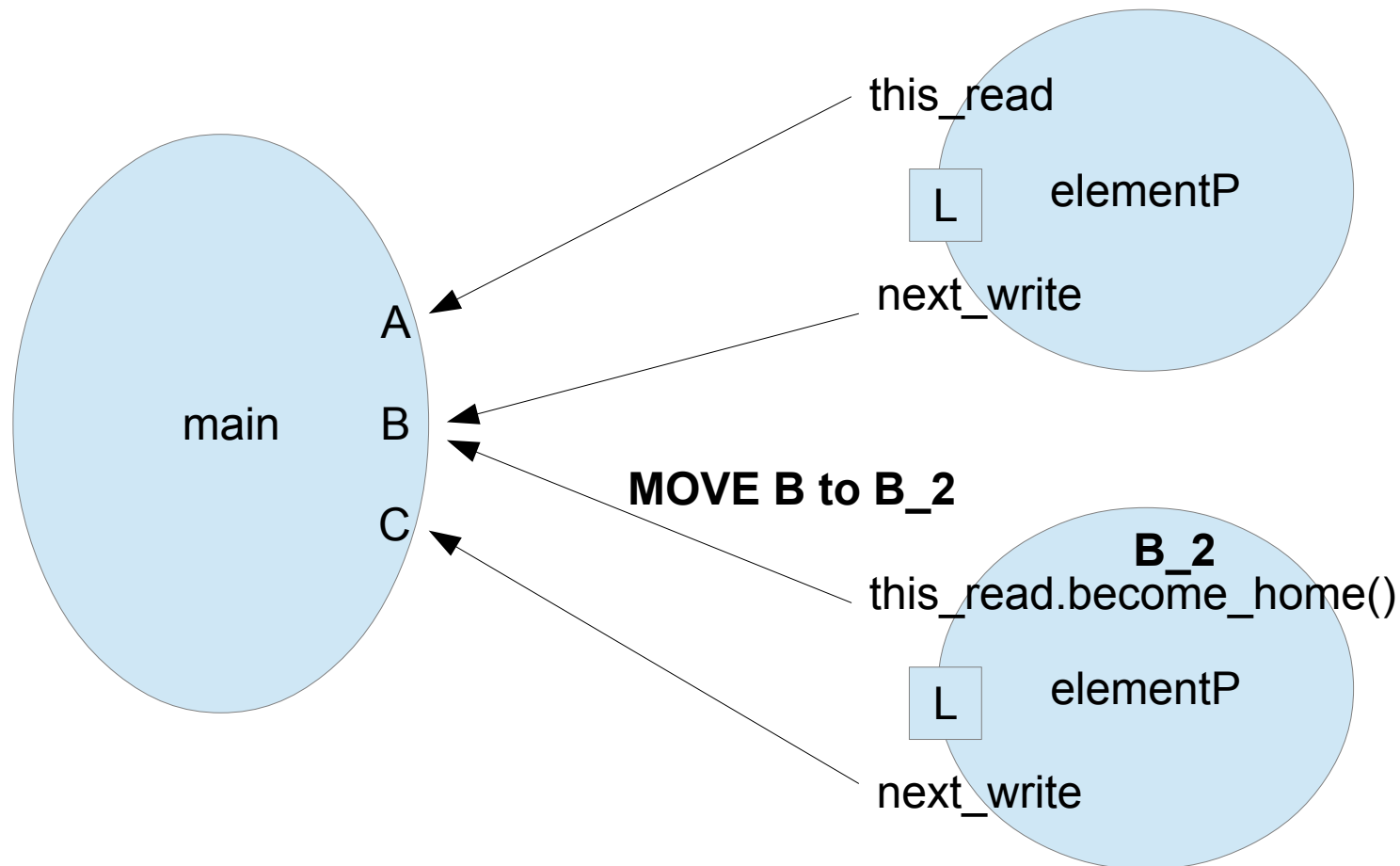
Introducing Mobile Channel Homes in PyCSP



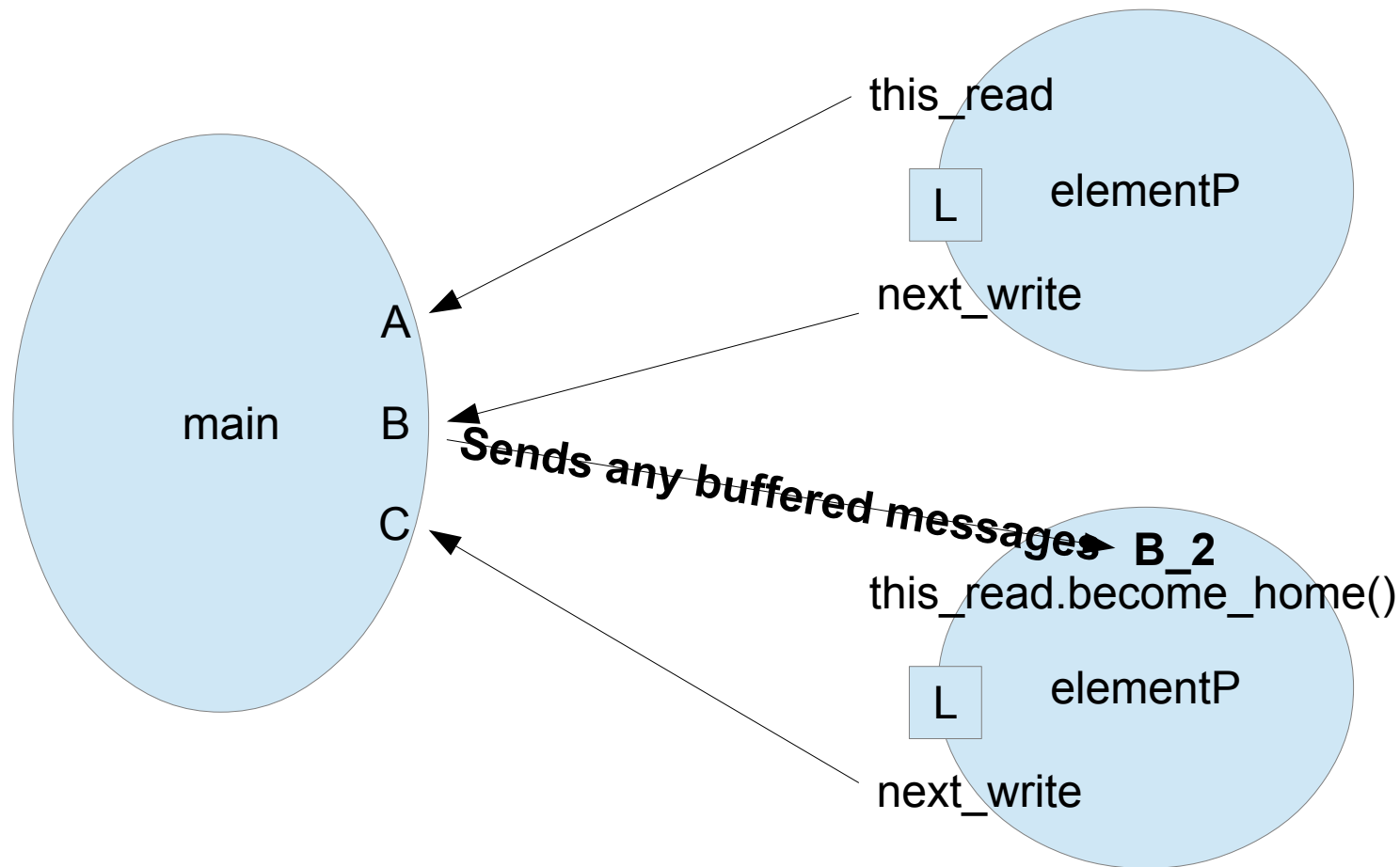
Introducing Mobile Channel Homes in PyCSP



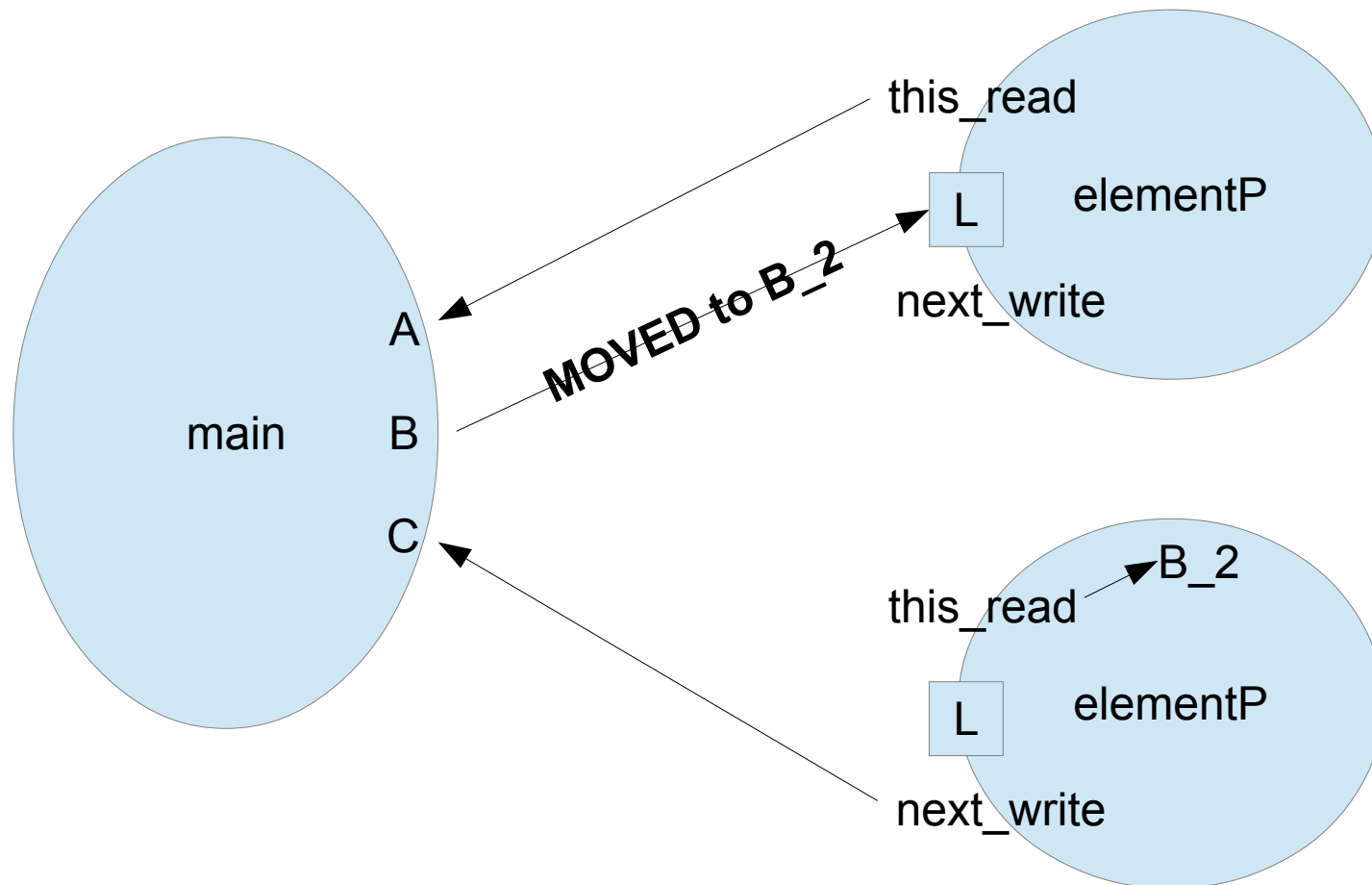
Introducing Mobile Channel Homes in PyCSP



Introducing Mobile Channel Homes in PyCSP

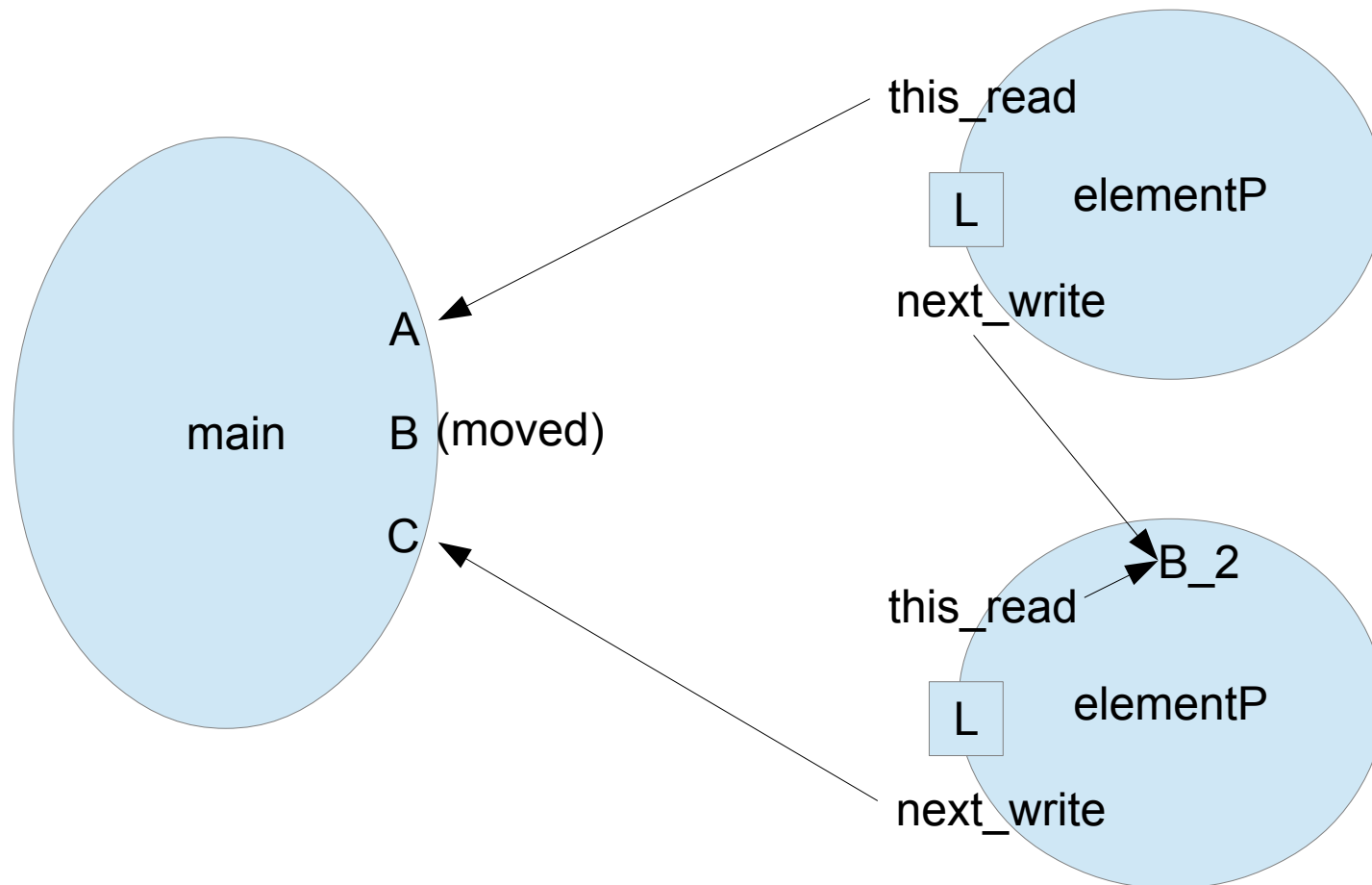


Introducing Mobile Channel Homes in PyCSP



B_2 is now the official channel home of B. If any processes connects to B at main, they will receive the message "MOVED to B_2"

Introducing Mobile Channel Homes in PyCSP

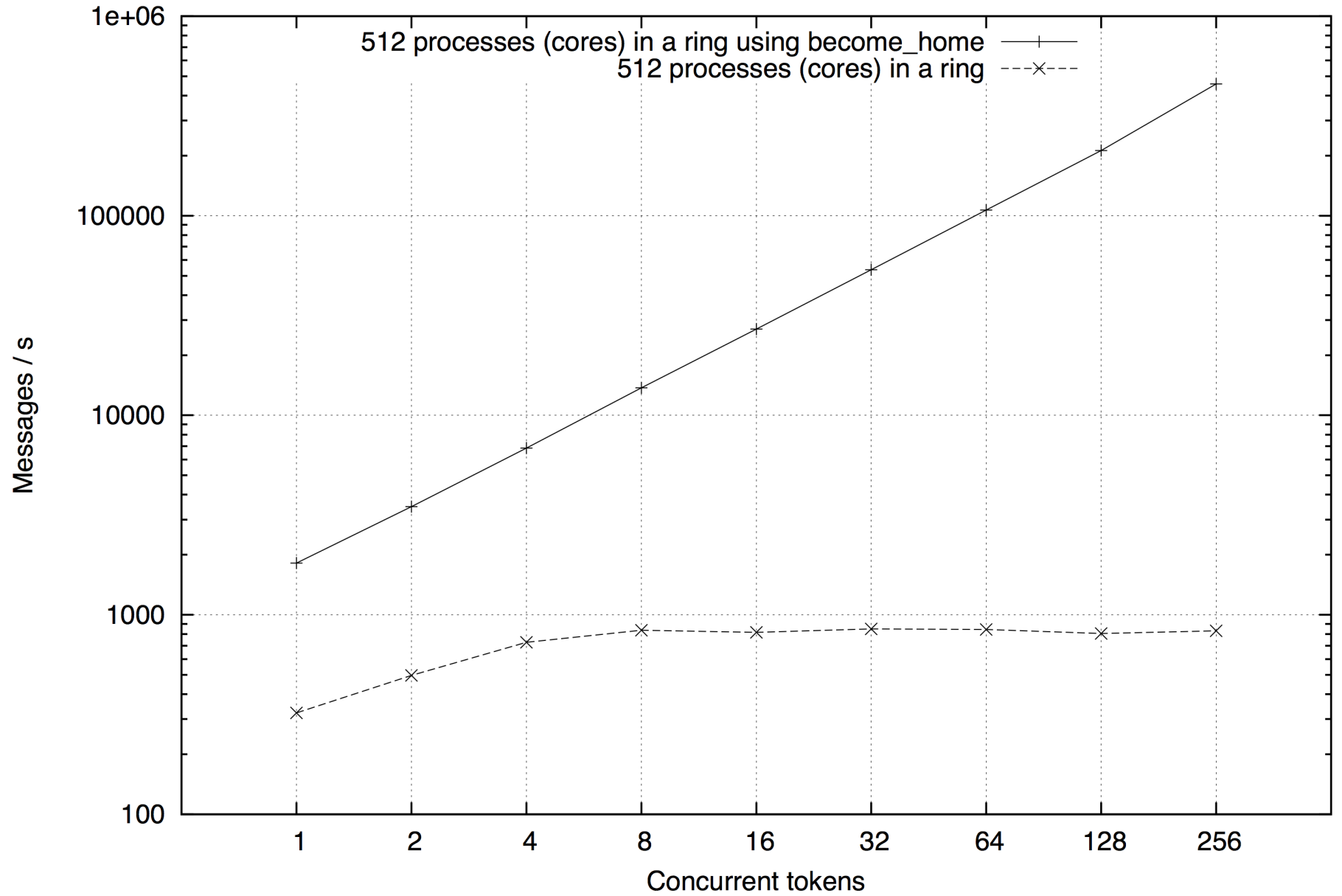


Introducing Mobile Channel Homes in PyCSP

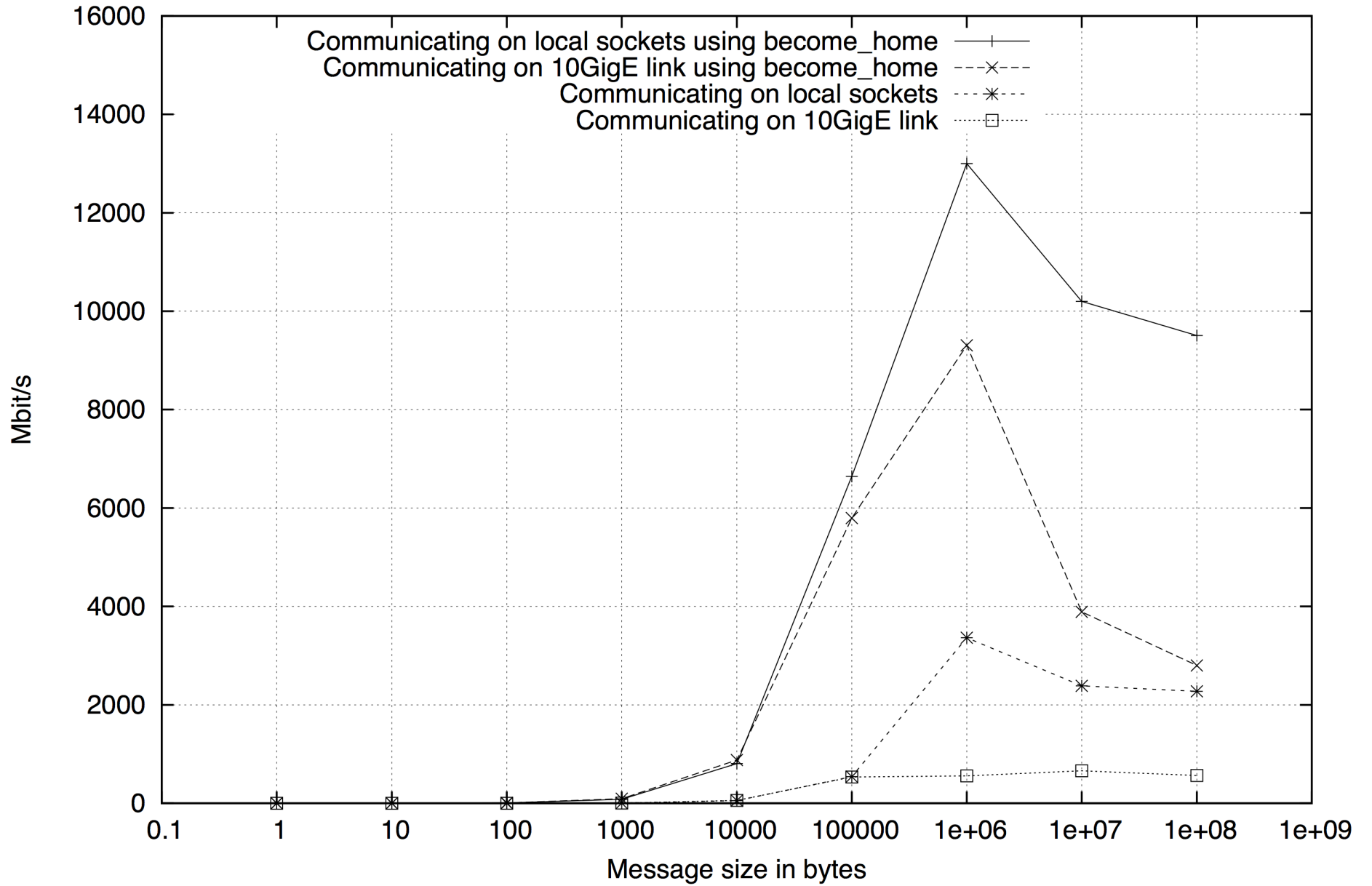
- The order of posted requests is not stable during a move of a channel home. Thus, priorities can not be guaranteed during this step.
- For most PyCSP applications, this is not expected to be an issue.

Results

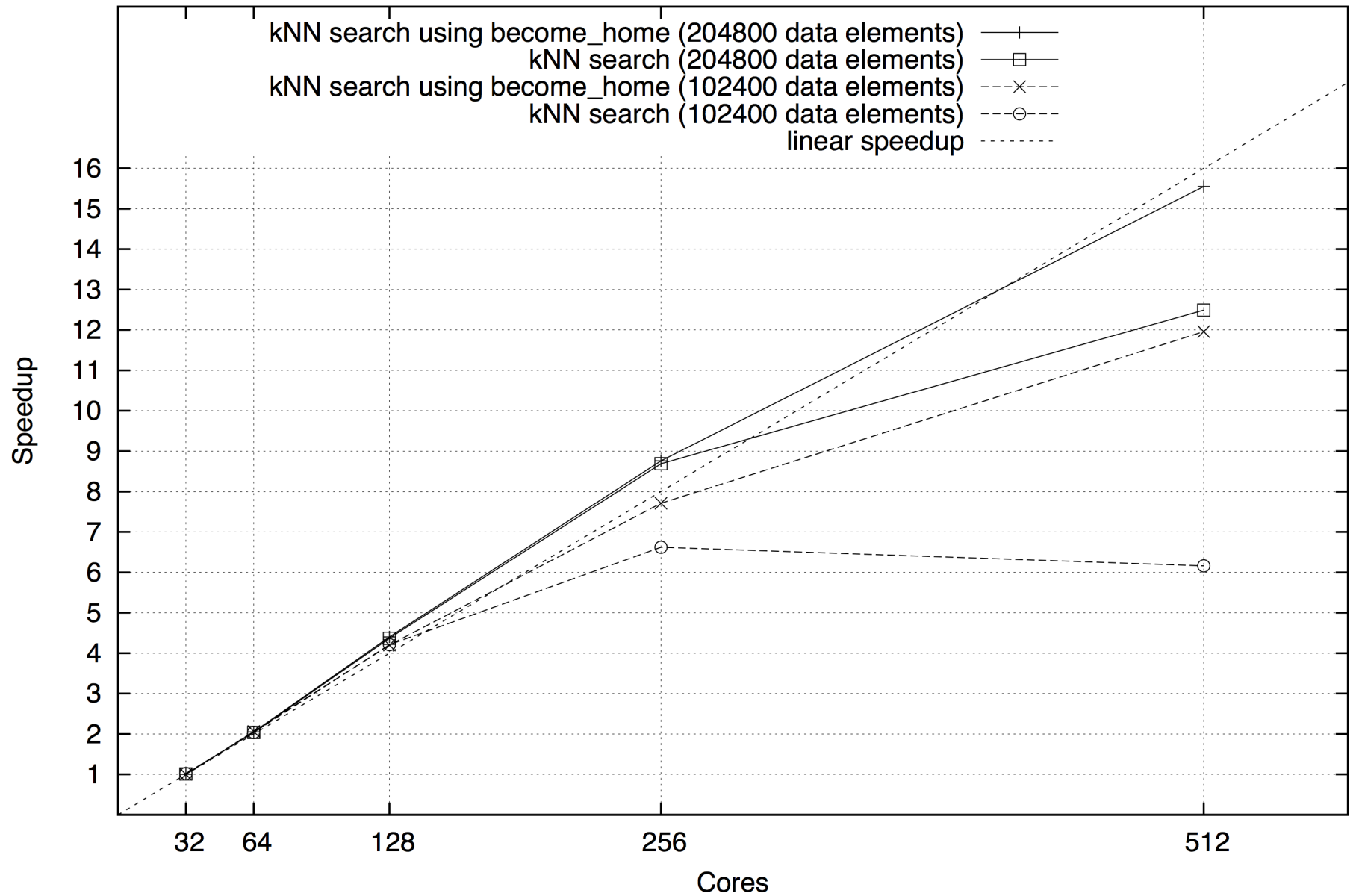
PyCSP tokenring benchmark



PyCSP throughput benchmark



PyCSP kNN benchmark



Conclusions

- PyCSP has distributed channels and remote processes, but only with the introduction of the `become_home()` functionality is PyCSP now able to scale seamlessly to large clusters
- With a few tweaks, we expect that PyCSP can scale beyond clusters with more than 512 cores.
- Thus, scientists does now have a tool which can handle large computations using CSP in a working Python environment.

Thanks