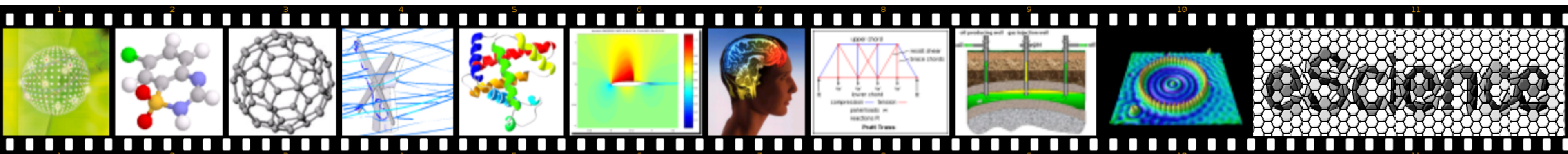
Three Unique Implementations of Processes for PyCSP

Rune M. Friborg, John M. Bjørndalen and Brian Vinter
eScience center, University of Copenhagen
University of Tromsø

PyCSP - before

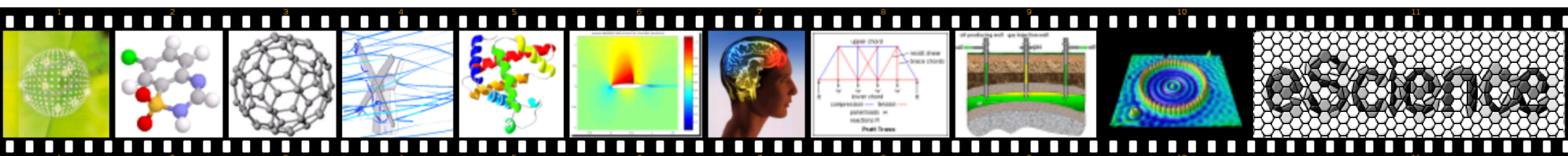
- CSP library for Python
- Every CSP process is a thread.
- Synchronization is handled by standard mutexes and conditions.
- Scheduling threads are handled by the operating system.



CPython – Global Interpreter Lock (GIL)

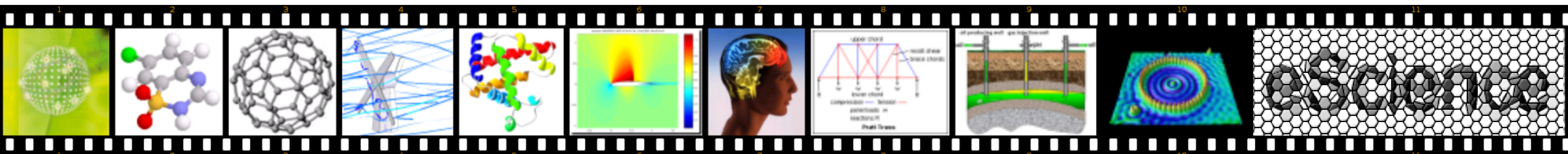
- CPython is the standard Python interpreter and is limited by the Global Interpreter Lock.
- This is the results of a simple Monte Carlo PI execution.

Workers	1	2	3	4	5
Threads	0.98s	1.52s	1.56s	1.55s	1.57s
Processes	1.01s	0.57s	0.54s	0.54s	0.56s



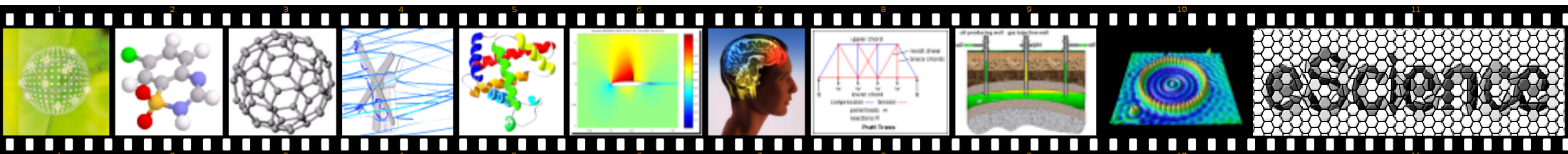
Why does the GIL issue matter?

- When prototyping we want to run processes in parallel without having to write an external module in C code.
- Compared to other CSP implementations it can be annoying that your nice, portable and simple python code, does not run in parallel when it just as easily could.



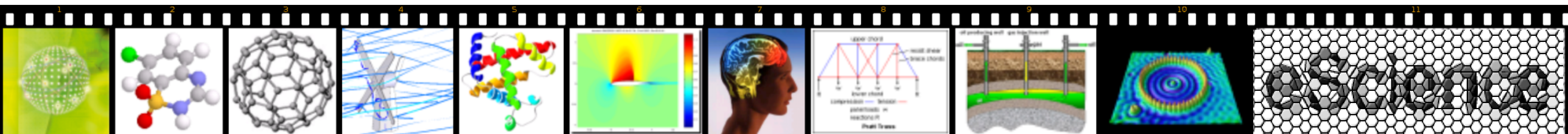
Why does the GIL issue matter?

- When prototyping we want to run processes in parallel without having to write an external module in C code.
- Compared to other CSP implementations it can be annoying that your nice, portable and simple python code, does not run in parallel when it just as easily could.



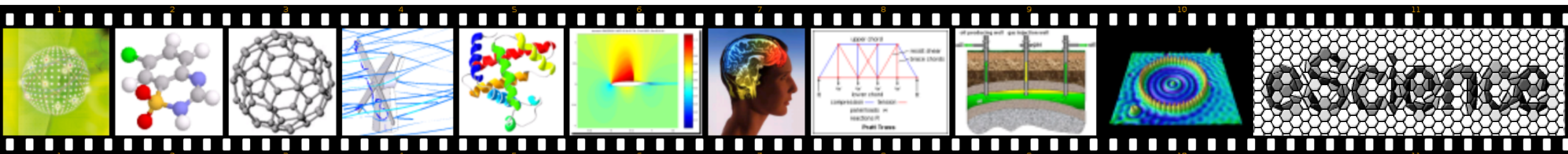
Limitation in number of threads

```
Exception in thread Thread-381:
Traceback (most recent call last):
  File "threading.py", line 460, in __bootstrap
    self.run()
  File "build/bdist.linux-i686/egg/pycsp/process.py", line 28,
in run
    self.retval = self.fn(*self.args, **self.kwargs)
  File "Sieve.py", line 18, in worker
    Spawn(worker(IN(child_channel), cout))
  File "build/bdist.linux-i686/egg/pycsp/process.py", line 38,
in Spawn
    _parallel(plist, False)
  File "build/bdist.linux-i686/egg/pycsp/process.py", line 50,
in _parallel
    p.start()
  File "threading.py", line 434, in start
    _start_new_thread(self.__bootstrap, ())
error: can't start new thread
```



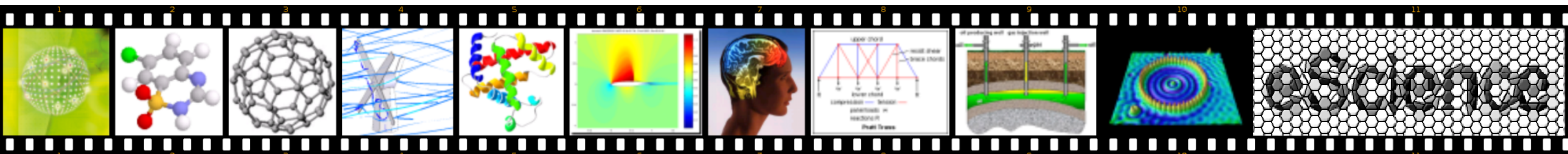
Why is this also a problem?

- Many CSP applications with a fine granularity of processes, does not make sense to implement in PyCSP, because of the overhead involved in communication and thread handling.



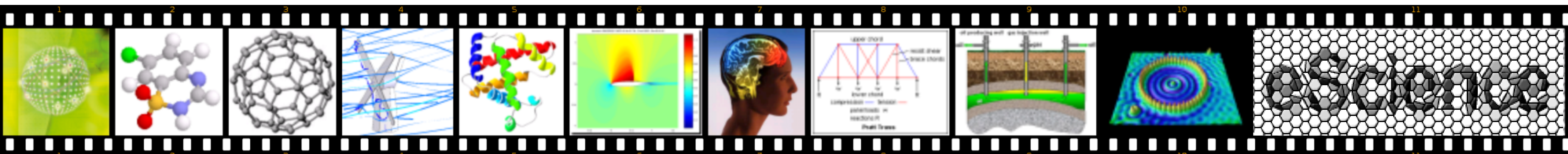
The Implementations

- All three implementations share a common API.
- Switching between them requires only changing the imported library.
  - Threads
    import pycsp.threads as pycsp
  - Processes
    import pycsp.processes as pycsp
  - Greenlets (co-routines)
    import pycsp.greenlets as pycsp
- The only changes made to the API from the PyCSP before is the addition of a @io decorator function.



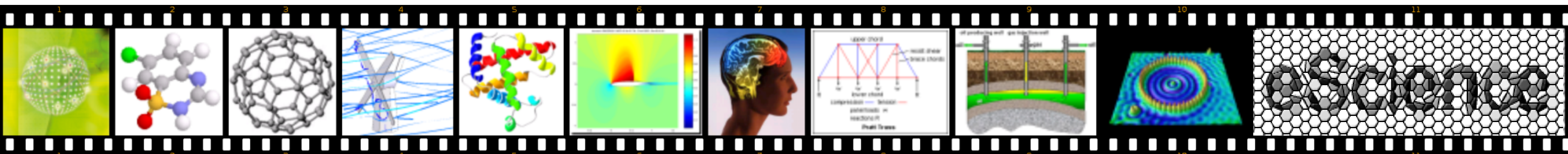
pycsp.processes

- Use the python multiprocessing module, that provides an API similar to the threading module.
- For OS processes we need to specifically define shared values, mutexes and conditions to communicate.
- The fork system call is simulated on the Windows system by the multiprocessing module.
- Data is copied when communicated. The communicated data is serialized and unserialized, thus references from external modules is required to support the 'pickle' module.
- The total amount of references to shared memory using the multiprocessing module is limited by the maximum number of open file descriptors per. process. To reduce the usage we have used a pool of shared mutexes and conditions.



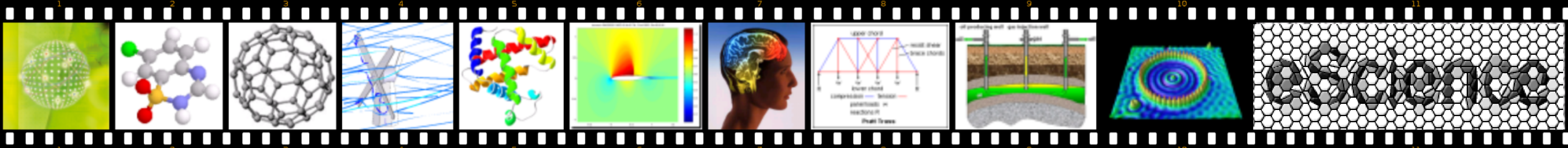
pycsp.processes

- Use the python multiprocessing module, that provides an API similar to the threading module.
- For OS processes we need to specifically define shared values, mutexes and conditions to communicate.
- The fork system call is simulated on the Windows system by the multiprocessing module.
- Data is copied when communicated. The communicated data is serialized and unserialized, thus references from external modules is required to support the 'pickle' module.
- The total amount of references to shared memory using the multiprocessing module is limited by the maximum number of open file descriptors per. process. To reduce the usage we have used a pool of shared mutexes and conditions.



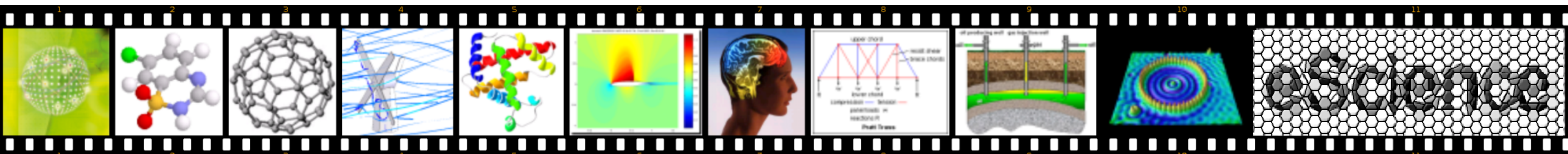
pycsp.processes

- Use the python multiprocessing module, that provides an API similar to the threading module.
- For OS processes we need to specifically define shared values, mutexes and conditions to communicate.
- The fork system call is simulated on the Windows system by the multiprocessing module.
- Data is copied when communicated. The communicated data is serialized and unserialized, thus references from external modules is required to support the 'pickle' module.
- The total amount of references to shared memory using the multiprocessing module is limited by the maximum number of open file descriptors per. process. To reduce the usage we have used a pool of shared mutexes and conditions.



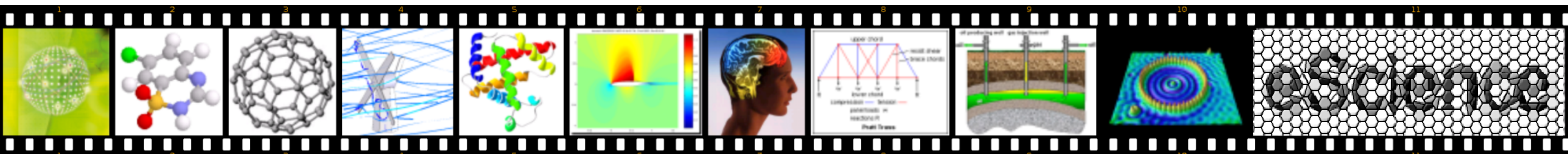
pycsp.processes

- Use the python multiprocessing module, that provides an API similar to the threading module.
- For OS processes we need to specifically define shared values, mutexes and conditions to communicate.
- The fork system call is simulated on the Windows system by the multiprocessing module.
- Data is copied when communicated. The communicated data is serialized and unserialized, thus references from external modules is required to support the 'pickle' module.
- The total amount of references to shared memory using the multiprocessing module is limited by the maximum number of open file descriptors per. process. To reduce the usage we have used a pool of shared mutexes and conditions.



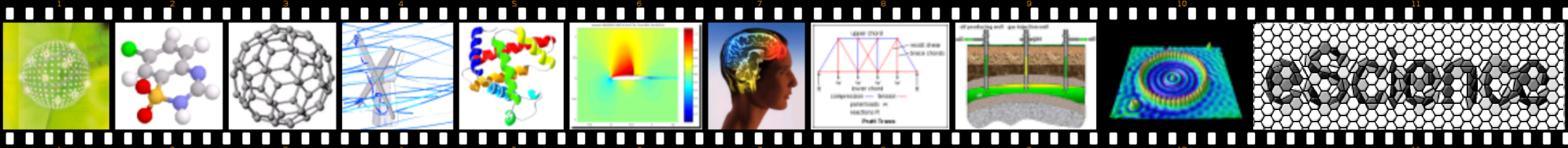
pycsp.processes

- Use the python multiprocessing module, that provides an API similar to the threading module.
- For OS processes we need to specifically define shared values, mutexes and conditions to communicate.
- The fork system call is simulated on the Windows system by the multiprocessing module.
- Data is copied when communicated. The communicated data is serialized and unserialized, thus references from external modules is required to support the 'pickle' module.
- The total amount of references to shared memory using the multiprocessing module is limited by the maximum number of open file descriptors per. process. To reduce the usage we have used a pool of shared mutexes and conditions.



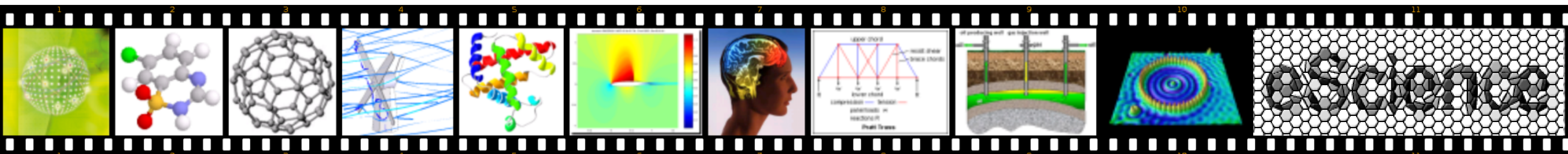
pycsp.processes

- Finally a memory allocator using the next-fit strategy was implemented to allow the communication of any data size. This was necessary, because all references to shared memory must be known on process start. Thus no dynamic allocation of shared memory using the multiprocessing module.



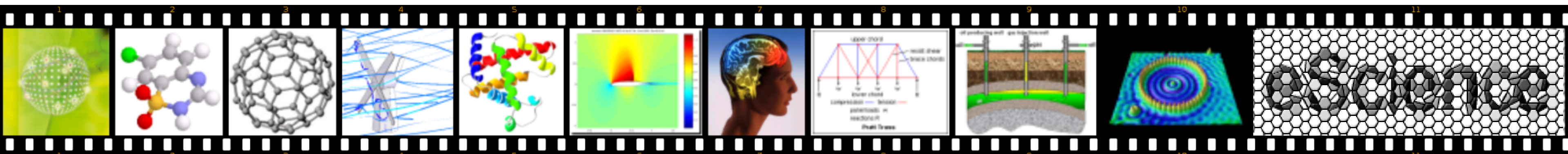
pycsp.greenlets

- Uses the external greenlet module. It allows us to control execution explicitly by calling the switch method.
- Small memory footprint compared to threads and processes.
- A simple FIFO scheduler controls the synchronization between greenlets.
- Blocking system calls?



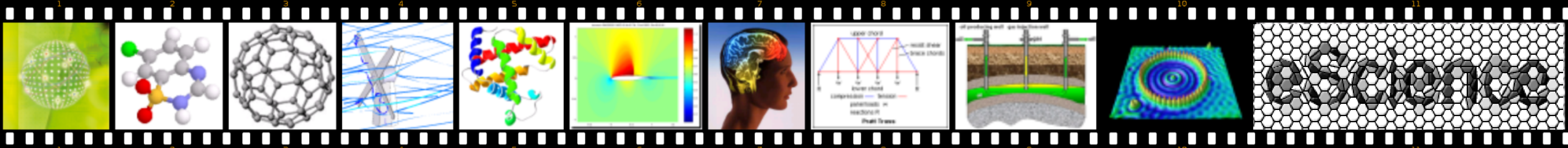
pycsp.greenlets

- Uses the external greenlet module. It allows us to control execution explicitly by calling the switch method.
- Small memory footprint compared to threads and processes.
- A simple FIFO scheduler controls the synchronization between greenlets.
- Blocking system calls?



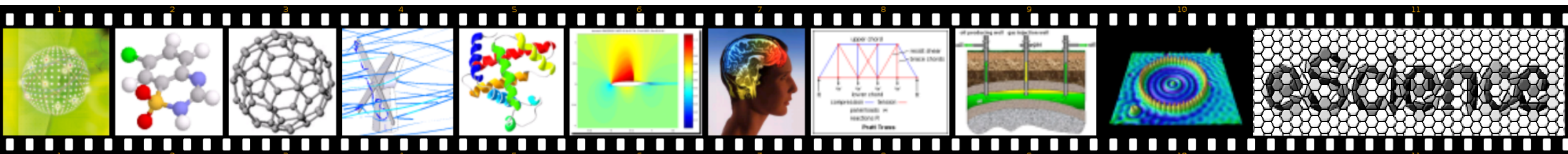
pycsp.greenlets

- Uses the external greenlet module. It allows us to control execution explicitly by calling the switch method.
- Small memory footprint compared to threads and processes.
- A simple FIFO scheduler controls the synchronization between greenlets.
- Blocking system calls?



pycsp.greenlets

- Uses the external greenlet module. It allows us to control execution explicitly by calling the switch method.
- Small memory footprint compared to threads and processes.
- A simple FIFO scheduler controls the synchronization between greenlets.
- Blocking system calls?



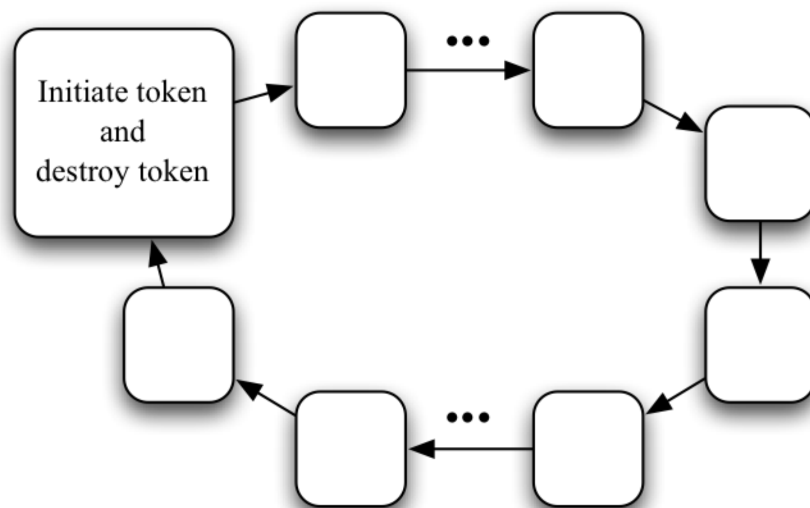
pycsp.greenlets - Yielding on Blocking IO

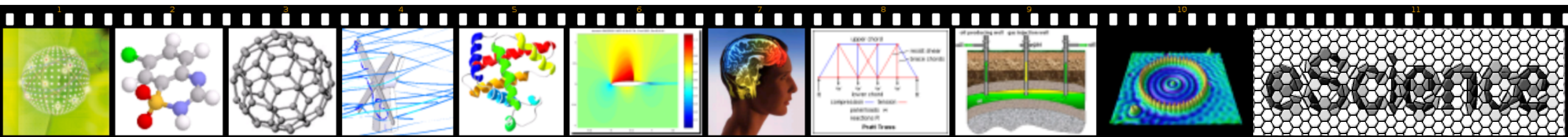
```
@io
def wait(seconds):
    time.sleep(seconds)

@process
def delay_output(msg, seconds):
    wait(seconds)
    print msg

Parallel(
  [delay_output('%d second delay' % i, i) for i in range(1, 11)]
)
```

Micro Benchmarks





One token in a ring of variable size

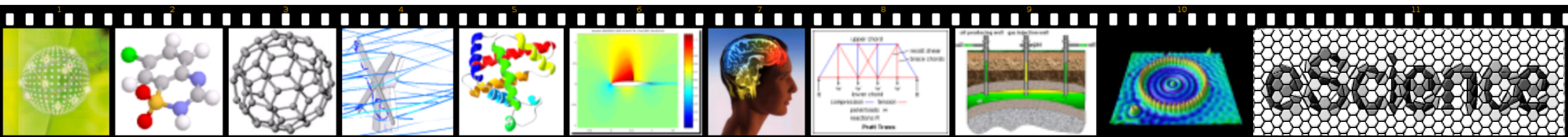
Chan. Comm. time (microseconds)

10000
1000
100
10

pycsp.threads
pycsp.processes
pycsp.greenlets

2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144

Ringsize (csp processes)



One to 63 concurrent tokens in a ring of 64 processes

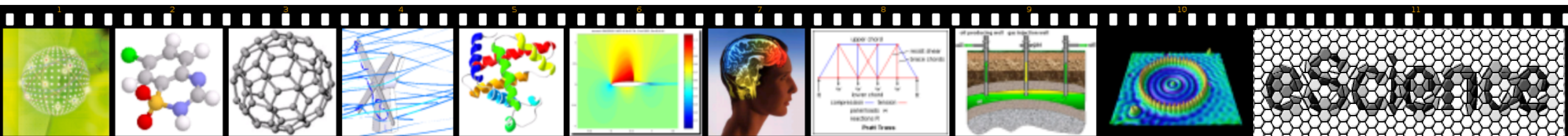
Chan. Comm. time (microseconds) / Tokens

350
300
250
200
150
100
50
0

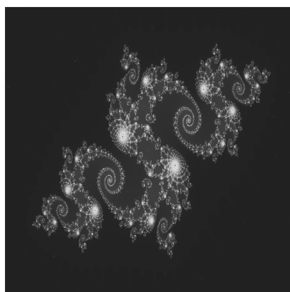
10 20 30 40 50 60

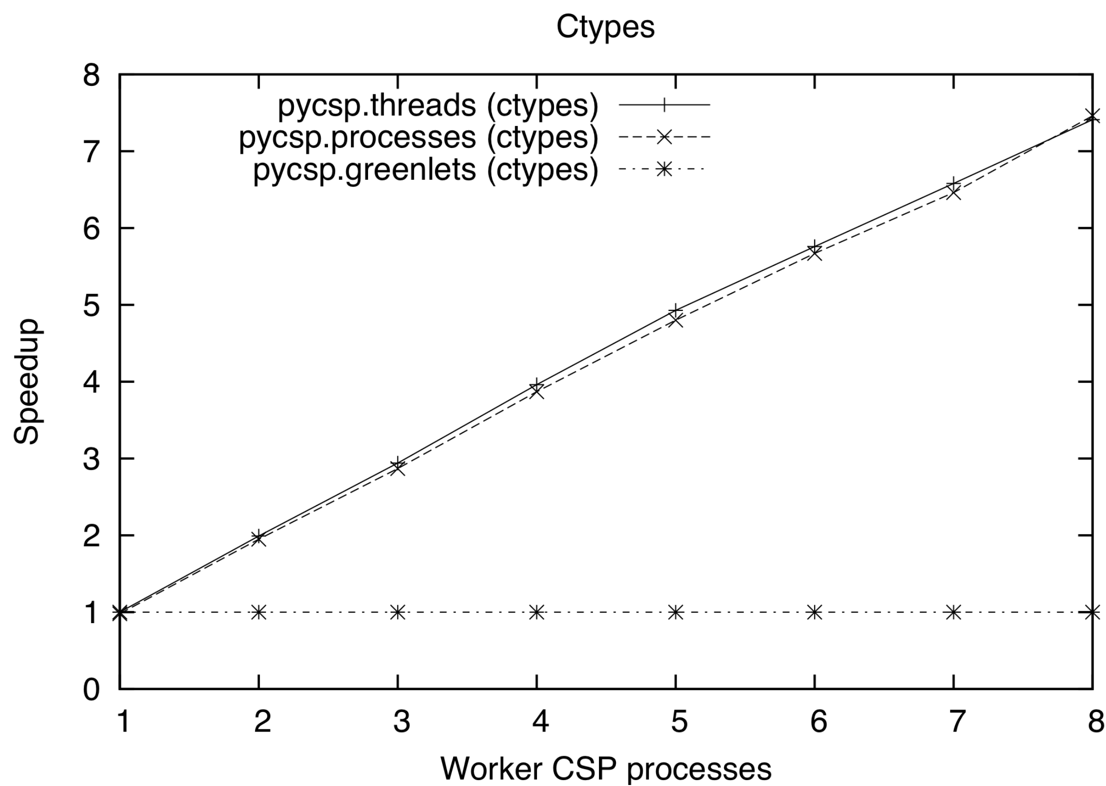
Concurrent Tokens

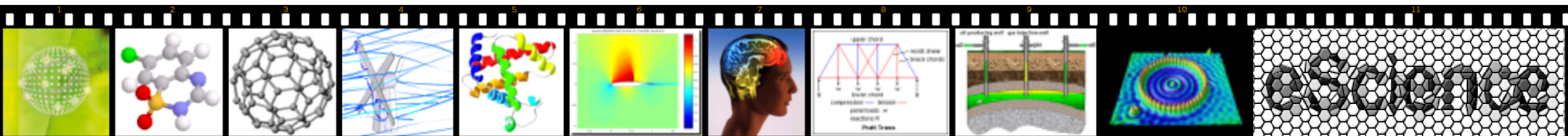
pycsp.threads
pycsp.processes
pycsp.greenlets



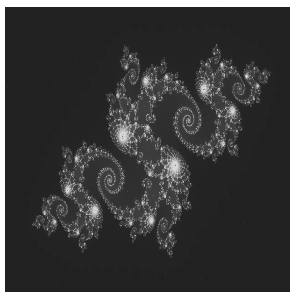
Mandelbrot - worker calling an external C module

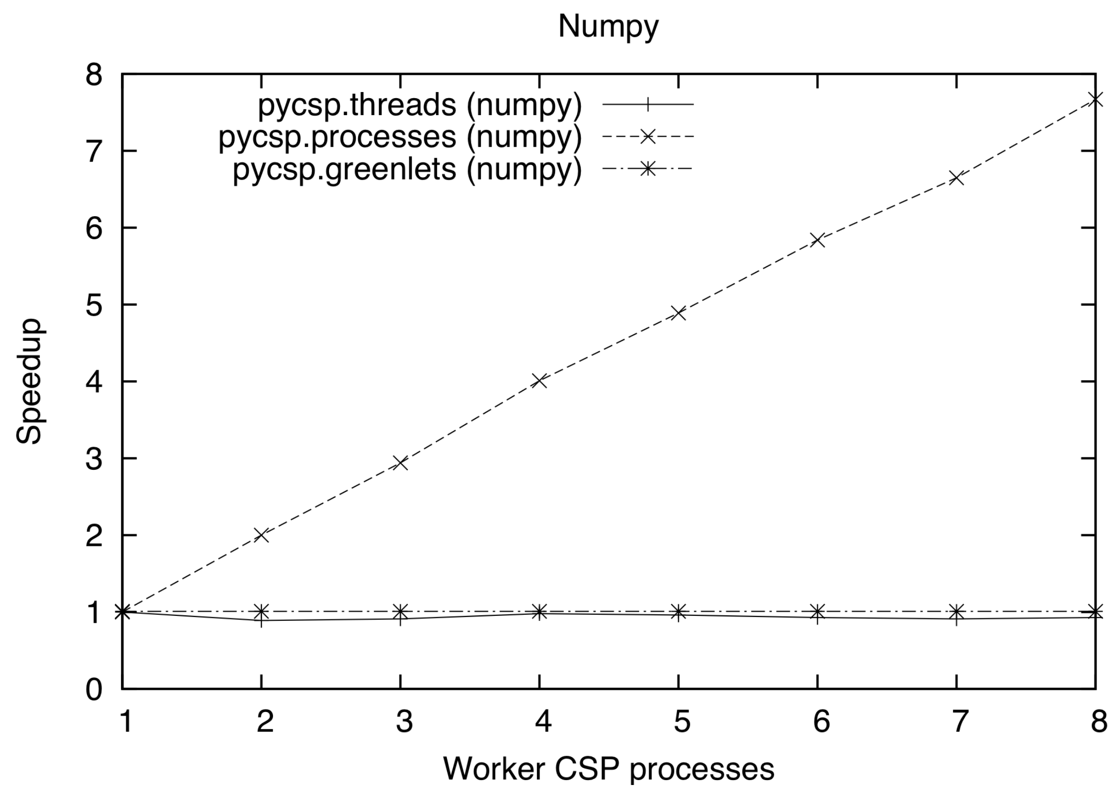




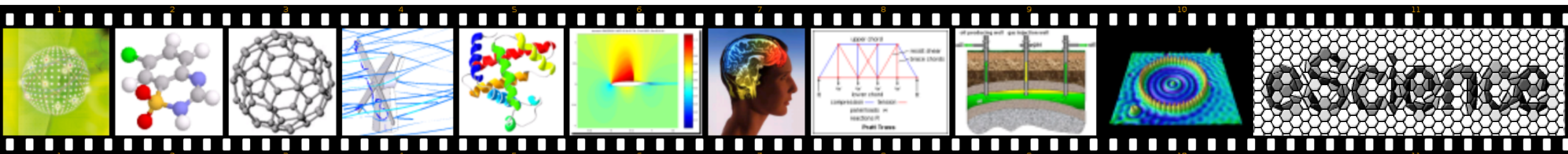


Mandelbrot – worker using only numpy



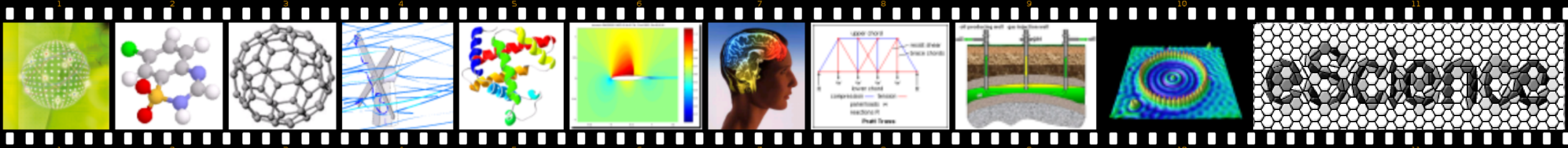


Finally we compare the advantages and limitations



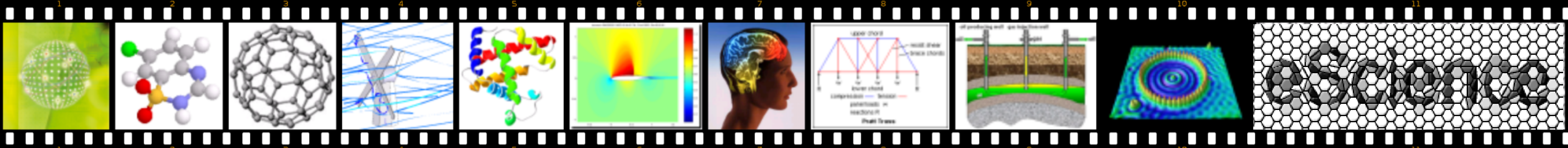
pycsp.threads

- Advantages
  - Only references to data are passed by channel communication.
  - Other Python modules usually only expect threads.
  - Compatible with all platforms supporting CPython 2.4+
- Limitations
  - Limited by the Global Interpreter Lock (GIL), resulting in very poor performance for code not releasing the GIL.
  - Limited in the maximum number of CSP processes possible.



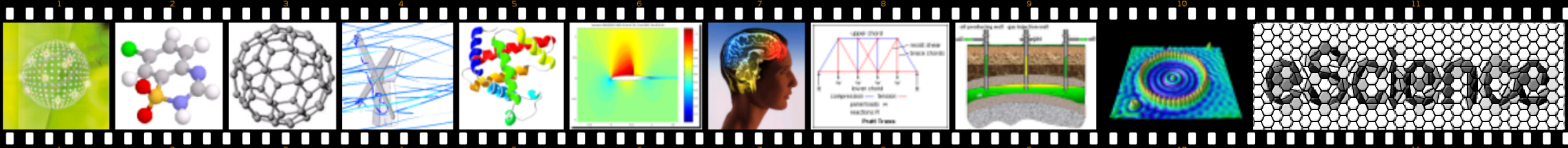
pycsp.processes

- Advantages
  - Can utilize more than one core, without requiring the developer to release the GIL.
  - All data communicated are serialized. The positive side-effect of serializing data is that data is copied when communicated, rendering it impossible to edit the received data from the sending process.
- Limitations
  - Fewer processes possible than pycsp.threads and pycsp.greenlets.
  - Windows support is limited, because of lack of the fork system call.
  - All data communicated are serialized, which requires the data type be supported by the pickle module.
  - Requires the python module 'multiprocessing' available in CPython 2.6+



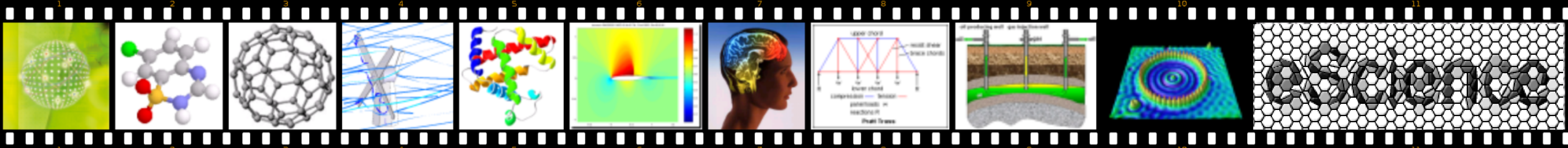
pycsp.greenlets

- Advantages
  - More optimal switching between CSP processes, since we can limit the context-switches to the point where they are blocking. Performance does not decrease with more CSP processes competing for execution.
  - Small footprint per CSP process, making it possible to run a larger number of processes, only limited by the amount of memory available.
  - Fast channel communications ($\approx$ 20μs).
- Limitations
  - No utilization of more than one CPU core.
  - Unfair execution, since execution is only yielded when a CSP process blocks on a channel.
  - Requires that the developer wraps blocking IO operations in an @io decorator to yield execution to another CSP process.
  - Requires installing the python module 'greenlet'.



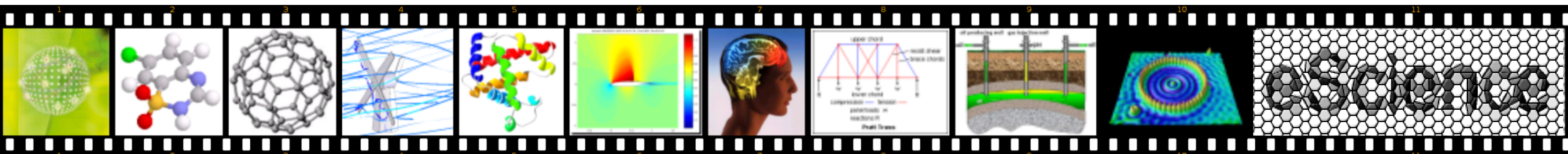
Conclusions

- Want 100.000 processes and prefers Python?
  - Use pycsp.greenlets
- Want parallism and prefers Python?
  - Use pycsp.processes
- Want it easy and prefers Python?
  - Use pycsp.threads



Future Work

- Making pycsp.threads run on the Android system.



Future Work

- ~~Making pycsp.threads run on the Android system~~.

5554:Test

1:07 PM

```
$ export TEMP="/sdcard/ase/extras/python/tmp/"
export PYTHONHOME="/data/data/com.google.ase/python"
export AP_PORT="48496"$ $
$ export PYTHONPATH="/sdcard/ase/extras/python/:/sdca
rd/ase/scripts/"
$ /data/data/com.google.ase/python/bin/python /sdcard
/ase/scripts/Sieve.py
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

Future Work

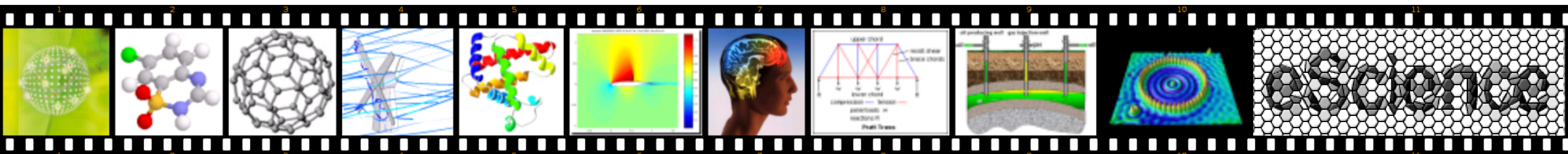
- pycsp.? – A distributed pycsp. Every channel is provided a name and is registered at a nameserver for lookups. The actual synchronization will be distributed.

Host 1:

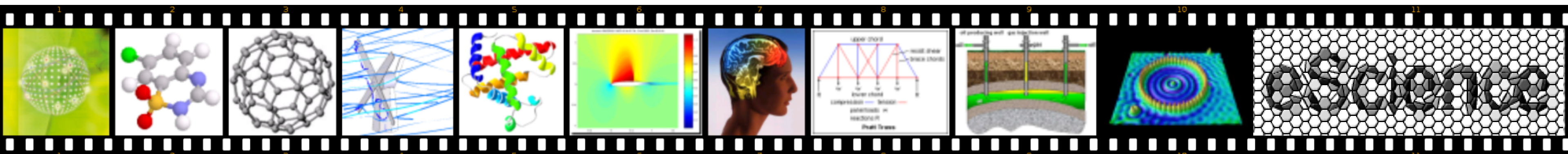
```
A = Channel('A')
Parallel(producer(A.writer()))
```

Host 2:

```
A = Channel('A')
Parallel(consumer(A.reader()))
```



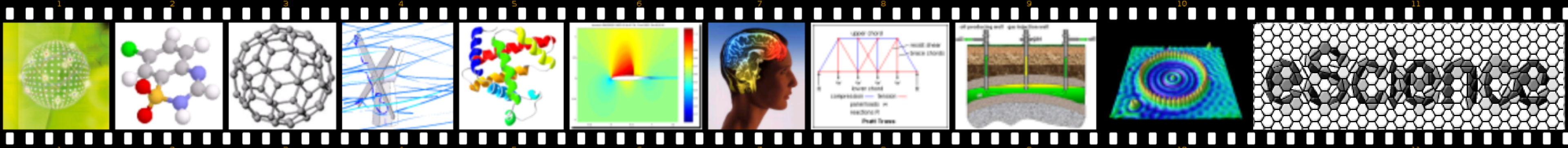
Questions?



Synchronization

```
def double_lock(req_1, req_2):
    if req_1.id < req_2.id:
        lock(req_1.lock)
        lock(req_2.lock)
    else:
        lock(req_2.lock)
        lock(req_1.lock)

for w in write_queue:
    for r in read_queue:
        double_lock(w, r)
        match(w, r)
        double_unlock(w, r)
```



Synchronization

Channel A

IR	OR
Active	Done, Fail
	Active

Channel B

IR	OR
Done, Ok	Done, Ok

Requests: 1, 2, 3, 4

1:
```
cin = A.reader()
id = cin()
```

2:
```
cout = A.writer()
cin = B.reader()
Alternation([
  { cin:received_id() },
  { (cout, id):send_id(id) }
]).execute()
```

3:
```
cout = A.writer()
cout(id)
```

4:
```
cout = B.writer()
cout(new_id)
```