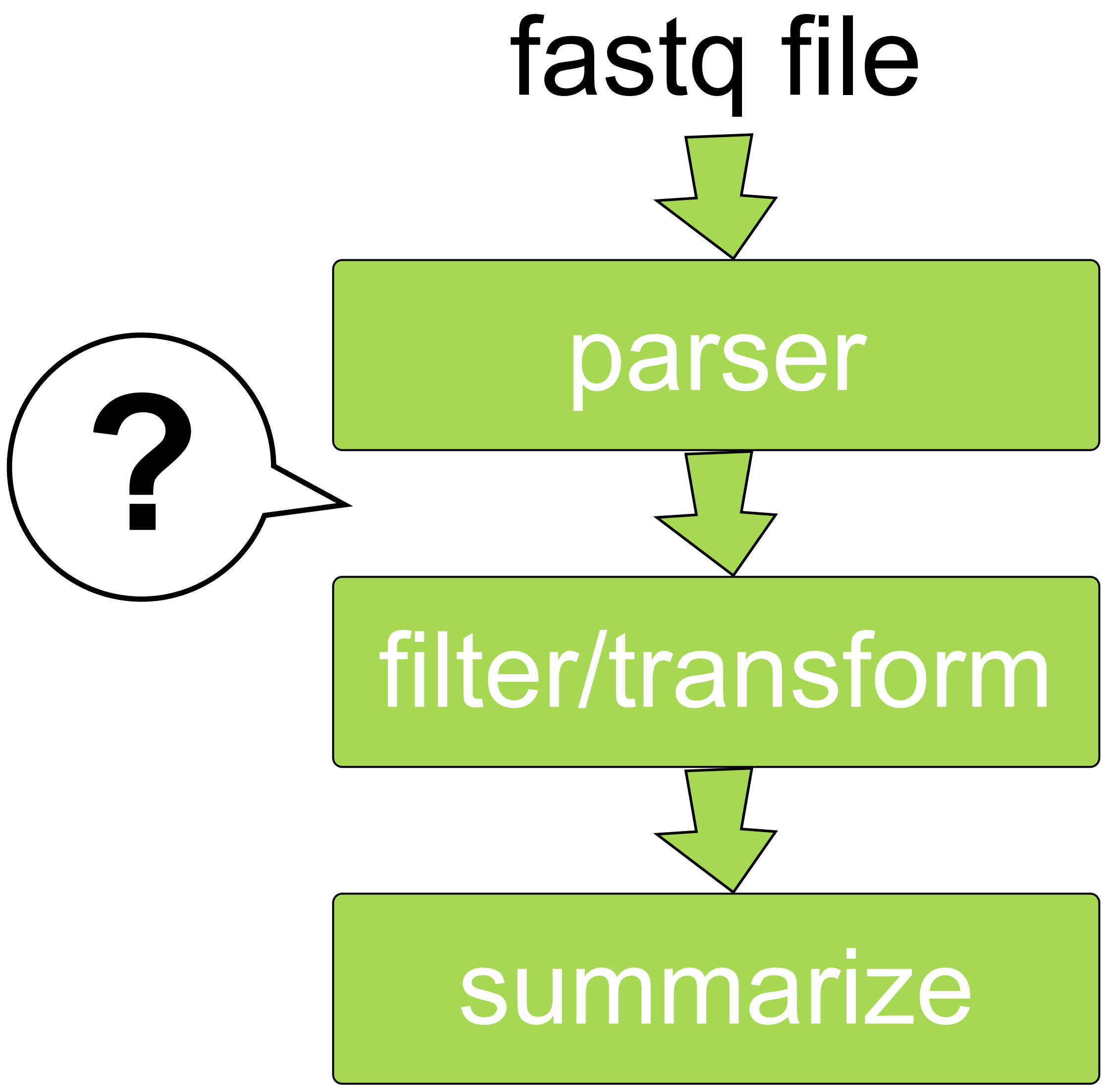
# [Parallel] processing pipelines in Python with generators/coroutines and multiprocessing

# Example: Fastq processing utility

- **Summaries:** Qualities, base composition, redundancy, ...
- **Filters:** read quality, uncalled bases, read length, ...
- **Transformations:** remove/collapse redundancy, change format, trim, modify quality scale

# sequential pipeline



## In what form is information moved through pipeline?

- tuple, list, dict
- python class
- python class with slots
- namedtuple
- cython cdef class

# speed and memory usage

Reading 2M tags into memory from fastq file (file size 272MB)

| | **memory [MB]** | **fold** | **time [s]** | **fold** |
|---|---|---|---|---|
| namedtuple | 470 | 1.4 | 12.6 | 1.3 |
| simple object | 720 | 2.2 | 20.9 | 2.2 |
| simple object with __slots__ | 457 | 1.4 | 12.4 | 1.3 |
| cdef class in cython with untyped __init__ and attributes | 457 | 1.4 | 9.1 | 0.9 |
| cdef class with c attributes | 328 | 1.0 | 9.6 | 1.0 |

# collections module

```
>>> import collections
>>> Tag = collections.namedtuple("Tag", "id seq qual")
>>> t=Tag("id1", "GATC", "bbbb")
>>> t
Tag(id='id1', seq='GATC', qual='bbbb')
>>> t=Tag("id1", "GATC", qual="bbbb")
>>> t.id
'id1'
>>> t[0]
'id1'
```

# Tag: subclass of namedtuple

```
class Tag(collections.namedtuple("Tag", "id seq qual")):
    """
    Lightweight, non-mutable object to pass along the pipeline
    stages
    """
    __slots__ = ()
    def __str__(self):
        return "@%s\n%s\n+\n%s" % self
    def __len__(self):
        return len(self.seq)
```

## component architecture

- use a component class that can keep state, receive input and produce output
- use generators or coroutines

# Python generators

```
def fibonacci(max_n):
    a, b = 0, 1
    while a <= max_n:
        yield a
        a, b = b, a + b


>>> fib_gen = fibonacci(8)
>>> for i in fib_gen:
...     print i,
... 0 1 1 2 3 5 8
```

generators produce a [potentially infinite] series of results using yield

calling a generator function returns a generator object but does not start execution

calling next() (i.e. iterating) starts/resumes execution

# Python coroutines

```
def filter_uncalled():
    while True:
        seqid, seq, qual = (yield)
        if seq.count(".") <= 1:
            print "@%s\n%s\n+\n%s" % (seqid, seq, qual)

In [2]: f = filter_uncalled()
In [3]: f
Out[3]: <generator object filter_uncalled at 0x1483288>
In [4]: f.next()
In [8]: f.send(("id1", "GATC", "bbbb"))
@id1
GATC
+
bbbb
In [9]: f.send(("id1", "GA..", "bbBB"))
<no output>
```

yield as an expression

“priming” the coroutine

# Fastq parser

```python
def fastq_to_tag(input):
    "transform iterable to Tags and pass them to target"
    for line in input:
        if line.startswith('@'):
            id1  = line.strip()[1:]
            seq  = input.next().strip()
            id2  = input.next().strip()[1:]
            qual = input.next().strip()
            if id2 != id1 or len(seq) != len(qual):
                raise ValueError, "error in input fastq format"
            yield Tag(id1, seq, qual)
```

# filters

```python
def filter_tags_with_uncalled_bases(input, max_allowed=1):
    for tag in input:
        if tag.seq.count(".") <= max_allowed:
            yield tag

def grep(input, motif):
    """motif is a string without ambiguity"""
    for tag in input:
        if motif in tag.seq:
            yield tag
```

# accumulators

```
def count_tags(input):
    return sum(1 for x in input)

def make_nr(input, trim5=0, trim3=0):
    """make nr after trimming trim3 nts of 3' end
    and trim5 nts of 5' end"""
    nr = collections.defaultdict(int)
    for tag in input:
        end = len(tag) - trim3
        nr[tag.seq[trim5:end]] += 1
    return nr
```

## example1: count let7a sequences

```
def let7a():
    """find tags that match let7a"""
    tags          = fastq_to_tag(open("test.fq"))
    no_uncalled = filter_tags_with_uncalled_bases(tags)
    let7a         = grep(no_uncalled, "TGAGGTAGTAGGTTGTATAGTT")
    let7a_count = count_tags(let7a)
    print "Found %i tags that match let7a sequence" % let7a_count
```

14s for 2M reads

# example 2: dinucleotide frequency at 5’ end of tags

```python
def dinucleotide():
    """frequency of dinucleotides at 5' end of reads"""
    tags         = fastq_to_tag(open("test.fq"))
    no_uncalled = filter_tags_with_uncalled_bases(tags, 0)
    full_length = filter_tags_by_len(no_uncalled, 36)
    nr          = make_nr(full_length, trim3=34)
    for freq, dinucleotide in sorted(((c, d) for d, c in nr.items()), reverse=True):
        print "%s: %8i" % (dinucleotide, freq)
```

# example 2: dinucleotide frequency at 5’ end of tags

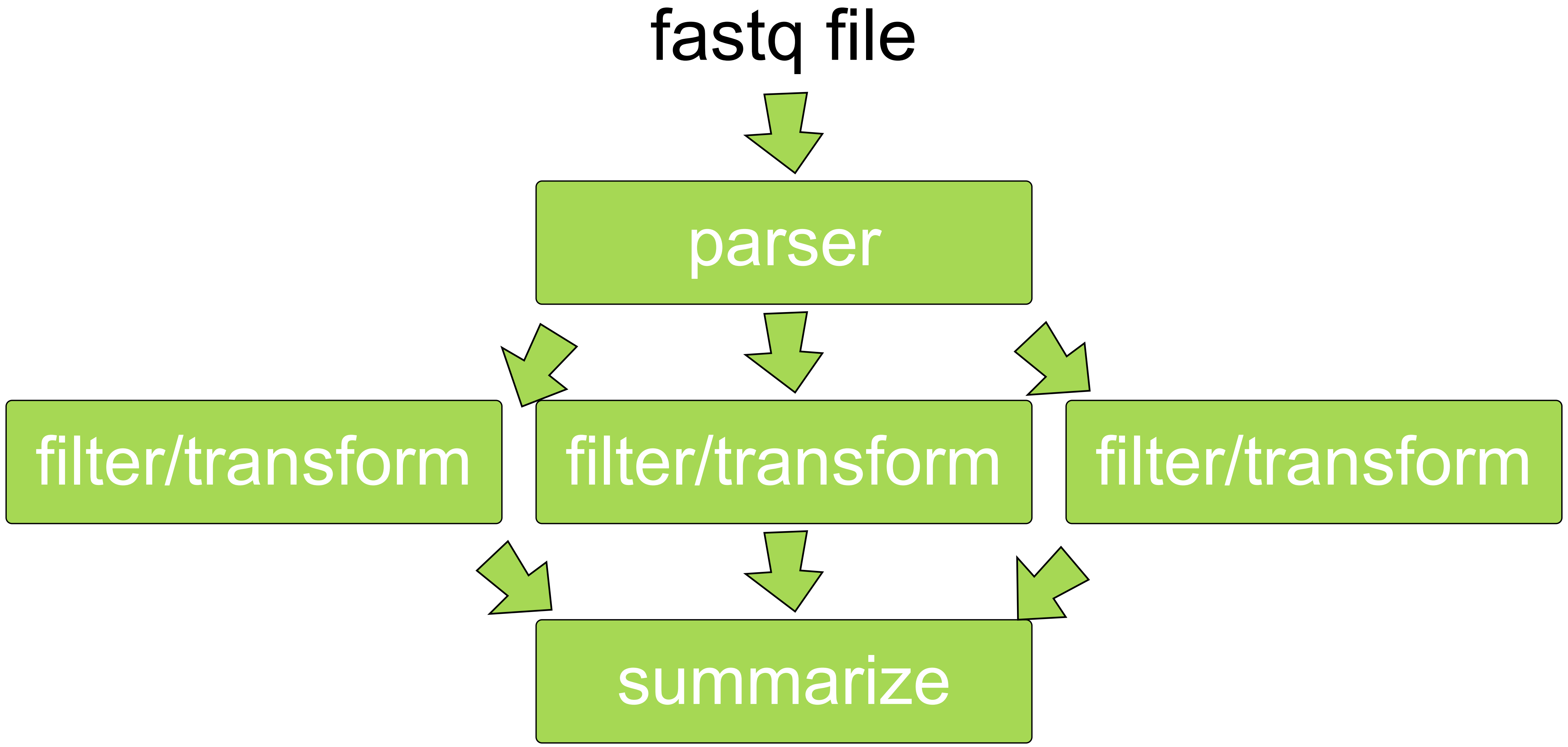
| | |
|---|---|
| GA: | 394591 |
| GG: | 292786 |
| AG: | 190706 |
| AA: | 148636 |
| CA: | 128992 |
| AC: | 122861 |
| GT: | 108033 |
| GC: | 106546 |
| CT: | 95235 |
| CC: | 90401 |
| AT: | 86794 |
| TG: | 85835 |
| CG: | 52560 |
| TT: | 32409 |
| TA: | 28906 |
| TC: | 21245 |

```
A = 25.7%
G = 25.6%
C = 23.7%
T = 25%
```

# example3: trim tags by quality

```
def phred64_to_p(s):
    return [10 ** (-0.1 * (ord(x) - 64)) for x in s]

def trim(input, min_len, min_prob):
    product = lambda a,b: a * b
    for tag in input:
        correct_prob = [1.0 - x for x in phred64_to_p(tag.qual)]
        tag_len = len(tag)
        left_coord = range(tag_len - min_len)
        right_coord = range(min_len, tag_len)
        possible_subsequences = [x for x in \
           itertools.product(left_coord, right_coord) \
           if x[1] - x[0] >= min_len]
        products = [(x[1] - x[0], \
           reduce(product, correct_prob[x[0]:x[1]]), x[0], x[1]) \
           for x in possible_subsequences]
        products.sort(reverse=True)
        for l, p, s, e in products:
            if p >= min_prob:
                yield Tag(tag.id, tag.seq[s:e], tag.qual[s:e])
                break
```

# parallel/distributed pipeline



## multiprocessing

"multiprocessing is a package that supports spawning processes using an API similar to the threading module"

# multiprocessing

```python
import multiprocessing as mp
import os

def info():
    print "  module:", __name__
    print "  parent:", os.getppid()
    print "  self:   ", os.getpid()

def f(name):
    print "Hello,", name
    info()

if __name__ == "__main__":
    print "Main line:"
    info()
    p = mp.Process(target=f, args=("wolf",))
    p.start()
    p.join()
```

```
--> python foo.py
Main line:
module: __main__
parent: 92856
self:   92884
Hello, wolf
module: __main__
parent: 92884
self:   92885
```

# task runner

```python
Task = collections.namedtuple("Task", "f iterable args kwargs")
def process_task(input_q, result_q):
    for task in iter(input_q.get, 'STOP'):
        try:
            result = task.f(task.iterable, *task.args, **task.kwargs)
        except:
            return
        # can't pickle generator
        if type(result) == types.GeneratorType:
            result = list(result)
        result_q.put(result)
```

## splitter 1

```
def proc_multi(f, input, nr_proc,
        chunksize=10000, args=[], kwargs={},
        returns_iter=False):
    task_queue = multiprocessing.Queue()
    result_queue = multiprocessing.Queue()

    # fire up processes
    for i in range(nr_proc):
        multiprocessing.Process(target=process_task,
                args=(task_queue, result_queue)).start()

    # feed the queue
    chunk = []
    submitted_tasks = 0
    for i in input:
        chunk.append(i)
        if len(chunk) == chunksize:
            task_queue.put(Task(f, chunk, args, kwargs))
            submitted_tasks += 1
            chunk = []
    if chunk != []:
        task_queue.put(Task(f, chunk, args, kwargs))
        submitted_tasks += 1
```

# splitter 2

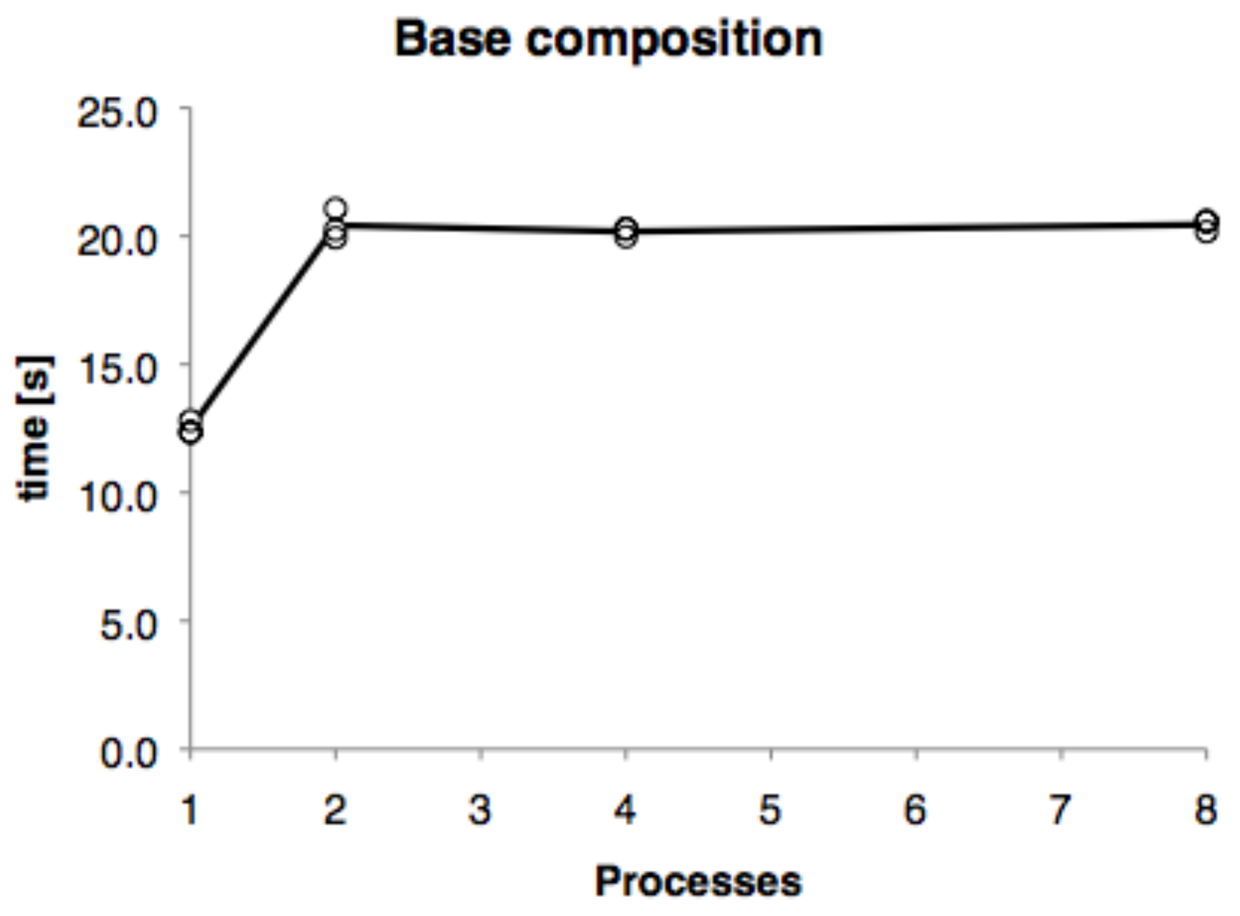
```
   # collect the results
    for i in range(submitted_tasks):
        result = result_queue.get()
        if returns_iter:
            try:
                for j in result:
                    yield j
            except TypeError:
                return
        else:
            yield result

    # tell processes to stop
    for i in range(nr_proc):
        task_queue.put('STOP')
```

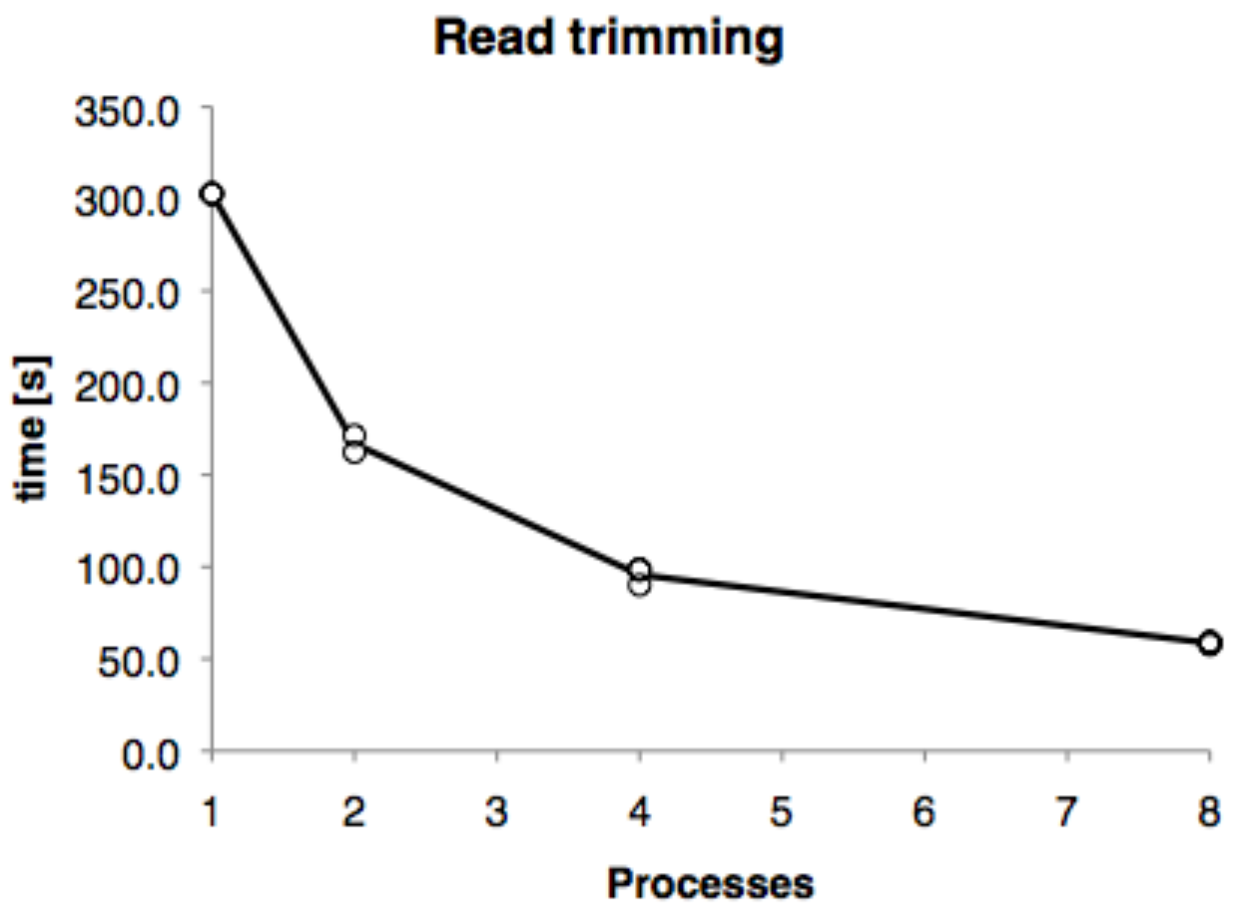
# example: trimming

```
def sequential2(filename="test_short.fq"):
    with open("trim.out", "w") as out:
        tags = fastq_to_tag(open(filename))
        trimmed_tags = trim(tags, min_len=25, min_prob=0.90)
        for tag in trimmed_tags:
            print >>out, tag
```

```
def parallel2(n=2, filename="test_short.fq"):
    with open("trim.out", "w") as out:
        tags = fastq_to_tag(open(filename))
        trimmed_tags = proc_multi(trim, tags, n,
                kwargs={"min_len": 25, "min_prob": 0.90},
                returns_iter=True)
        for tag in trimmed_tags:
            print >>out, tag
```

# message passing cost


Base composition
time [s]
Processes


Read trimming
time [s]
Processes