“This is a parallel parrot!” (CPA 2011 fringe edition)

Adam Sampson
Institute of Arts, Media and Computer Games
University of Abertay Dundee

Abertay University

Python

- Dynamically typed
- Multiparadigm
- Indentation-structured
- Designed to support teaching
- Widely deployed and used
- Lots of good-quality libraries
- **Really** slow bytecode interpreter

```
import sys, re

space_re = re.compile(r'[ \t\r\n\f\v]+')
punctuation_re = re.compile(r'[!"#%&\'()*,-./:;?@\[\\\]_{}]+')

max_words = int(sys.argv[1])
f = open(sys.argv[2])
data = f.read()
f.close()

words = [re.sub(punctuation_re, '', word).lower()
         for word in re.split(space_re, data)]
words = [word for word in words if word != ""]

found = {}
for i in range(len(words)):
    max_phrase = min(max_words, len(words) - i)
    for phrase_len in range(1, max_phrase + 1):
        phrase = " ".join(words[i:i + phrase_len])
        uses = found.setdefault(phrase, [])
        uses.append(i)

for (phrase, uses) in found.items():
    if len(uses) > 1:
        print ('<"%s":(%d,[%s])>'
               % (phrase, len(uses), ",".join(map(str, uses))))
print
```

Benchmarking

- Machine:
  - 2x 2.27GHz Intel E5520 – 8 cores, 16 HTs
  - 12GB RAM; files in cache for benchmarks
  - Debian etch x86_64 with Python 2.6.6
- Using WEB.txt, 3 words, output to /dev/null
- Concordance.hs: **(still waiting)**
- ConcordanceTH.hs: **22.7s**
- mini-concordance.py: **13.5s**

Abertay University

Parallel Python

- Python's had threading support for a long time
- … but the bytecode engine is single-threaded
  - The “Global Interpreter Lock”
- Useful for IO-bound programs, or where you're mostly calling into native code
- No good for parallelising pure-Python code

Multiprocessing

- The **multiprocessing** module provides the same API as the **threading** module...
- … but it uses operating system processes
- Synchronisation becomes more expensive, but you can execute in parallel

So let's parallelise...

- This is a **trivially-parallelisable** problem
  - You can break it down into separate jobs that don't need to interact with each other
- Split up input file into **C** chunks
- Do concordance on each in parallel
- Merge results from different chunks together
- Print them out

Split

- Pick **C** points in the file
- Seek to each point
- Read forward until you find a word boundary
- Read a few words more forward to handle overlap between chunks
- Don't have to read the whole file
- Cheap – O(**C**) – not worth parallelising

Concordance

- Read appropriate chunk of file and do concordance just as before
  - IO has been parallelised
- Return dict (hashed map) of phrases to uses, and number of words read in total
- Parallelise using **multiprocessing.Pool**

```
pool = Pool(processes=C)  # num to run at once

jobs = []
for i in range(C):
    jobs.append(pool.apply_async(concordance, (args ...)))

results = [job.get() for job in jobs]
```

Abertay University

Merge

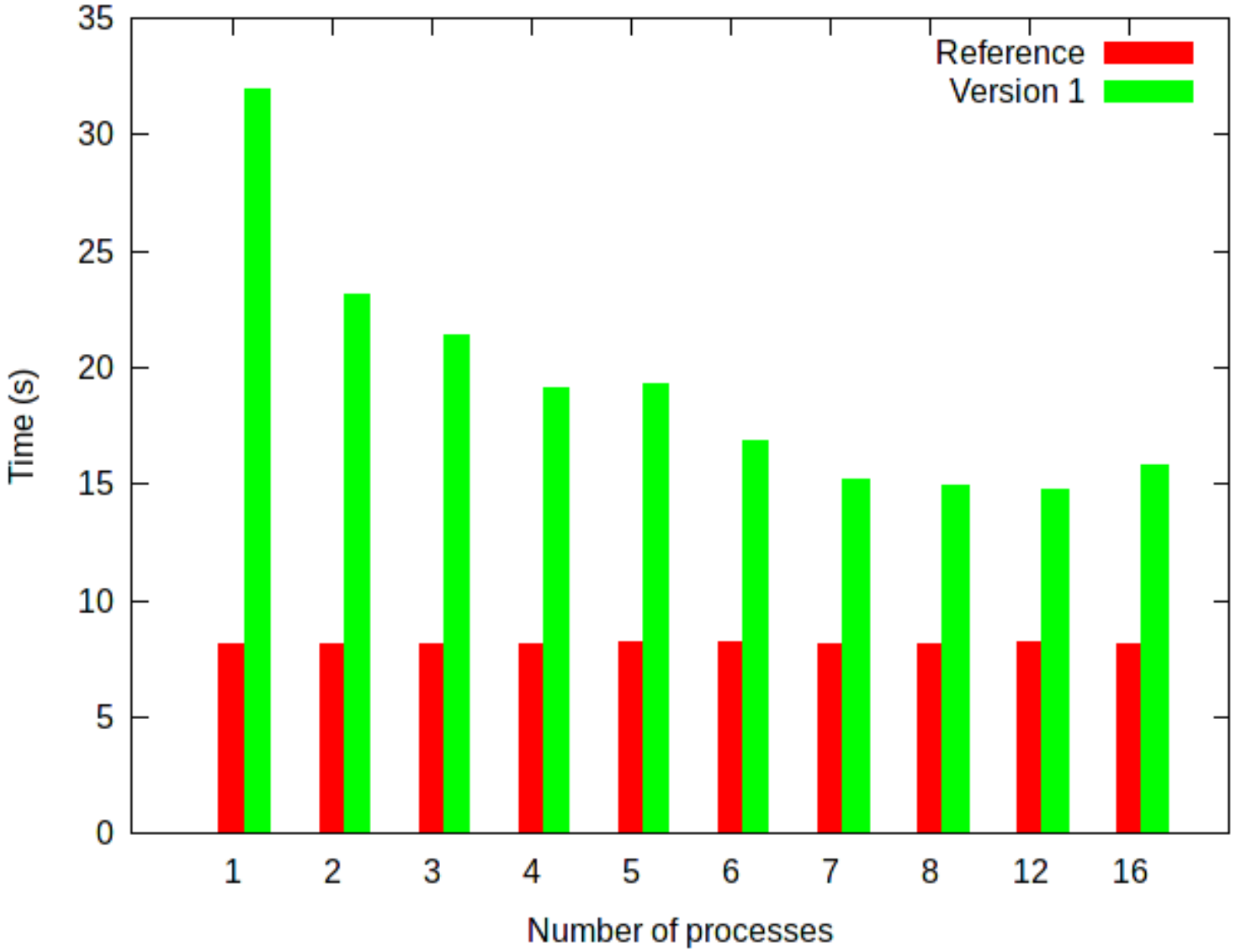
- Iterate through all the results, and add to a dict, adjusting word numbers based on the totals

```
merged = {}

first_word = 0
for (found, num_words) in results:
    for (phrase, uses) in found.items():
        all_uses = merged.setdefault(phrase, [])
        all_uses += [use + first_word for use in uses]
    first_word += num_words

return merged
```

Version 1

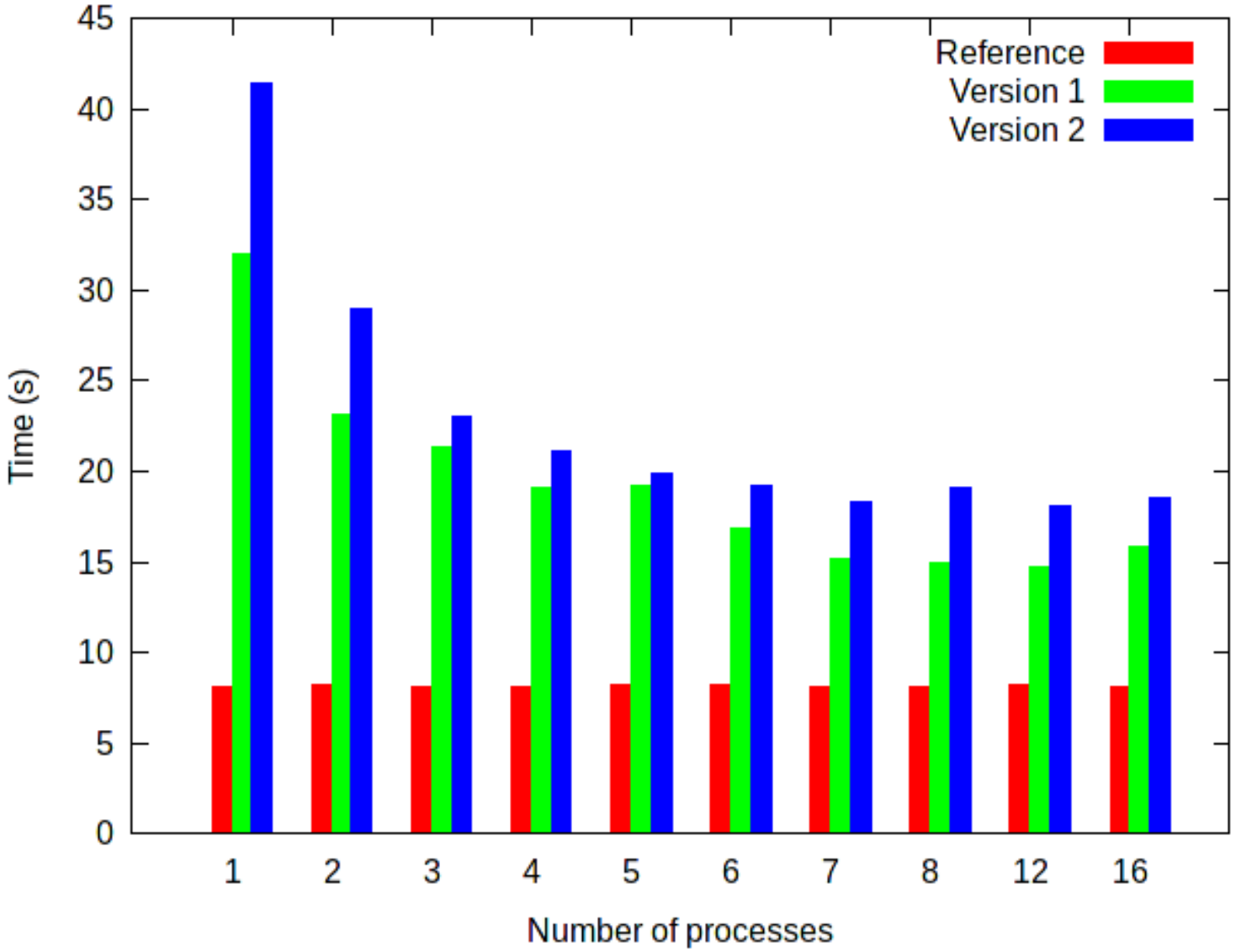


Abertay University

Hmm...

- Some scalability, but there's a massive constant overhead
- At this point, I forget Rule 3 of optimisation...
  - 1. Don't
    2. Don't yet
    3. Profile first
- The merge must be the slow part, right?
- Rewrite to sort in each concordance, and use **heapq.imerge** to merge sorted lists...

Version 2



Abertay University

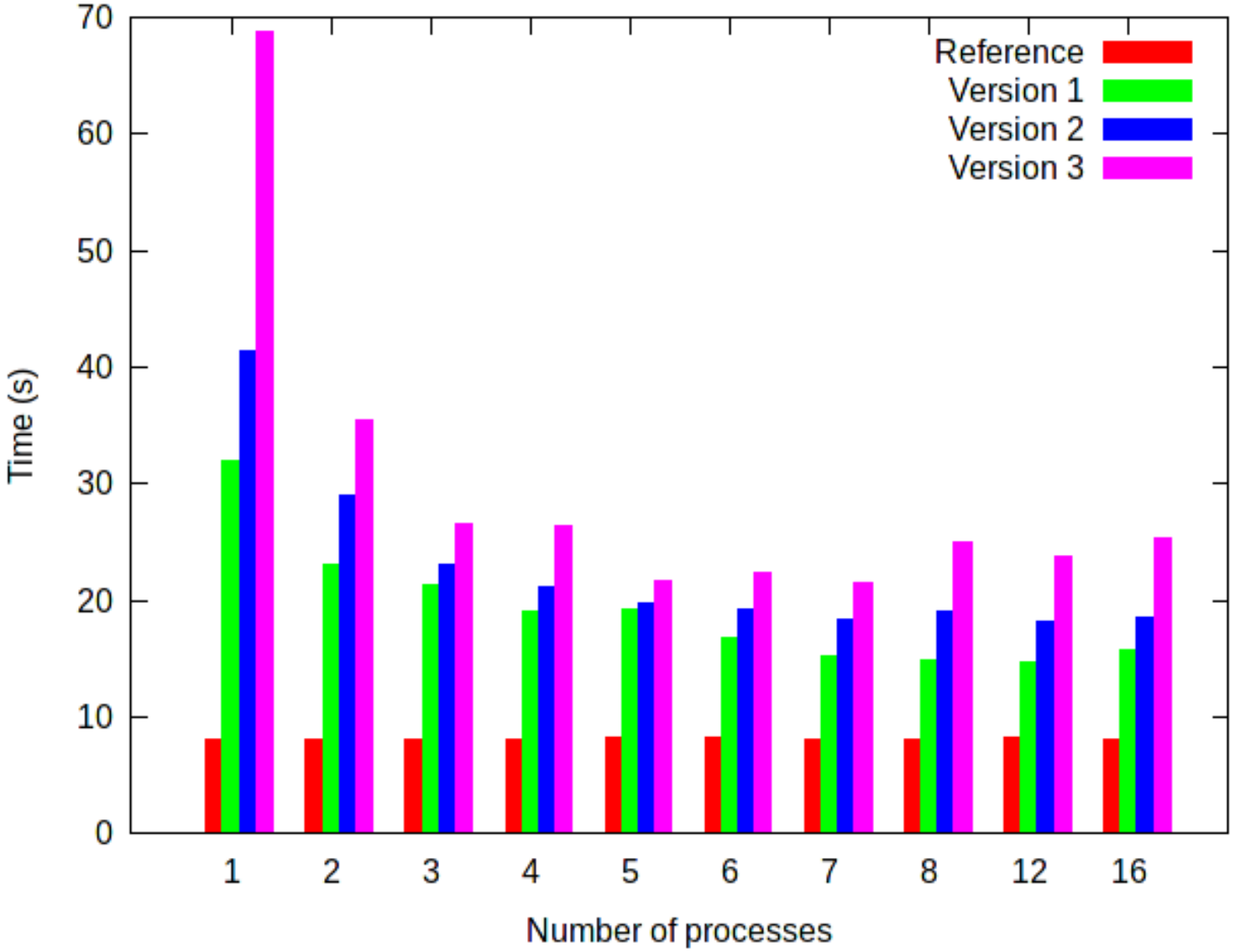
Well, that didn't work...

- Complicated Python is often slower...
  - ... because the runtime system and libraries are well-optimised for the common cases
- Stick with the obvious approach!
- Parallelise the merge instead

Parallel merge

- Compute hash of each phrase (Python **hash**), and group phrases by hash % **C**
- Each concordance returns several dicts
- Each merge takes all the dicts with the same hash % **C**, merges as before, and returns its merged dict
- Output iterates through merged dicts
  - It's useful that the output doesn't have to be sorted (although sorting the strings would be cheap)

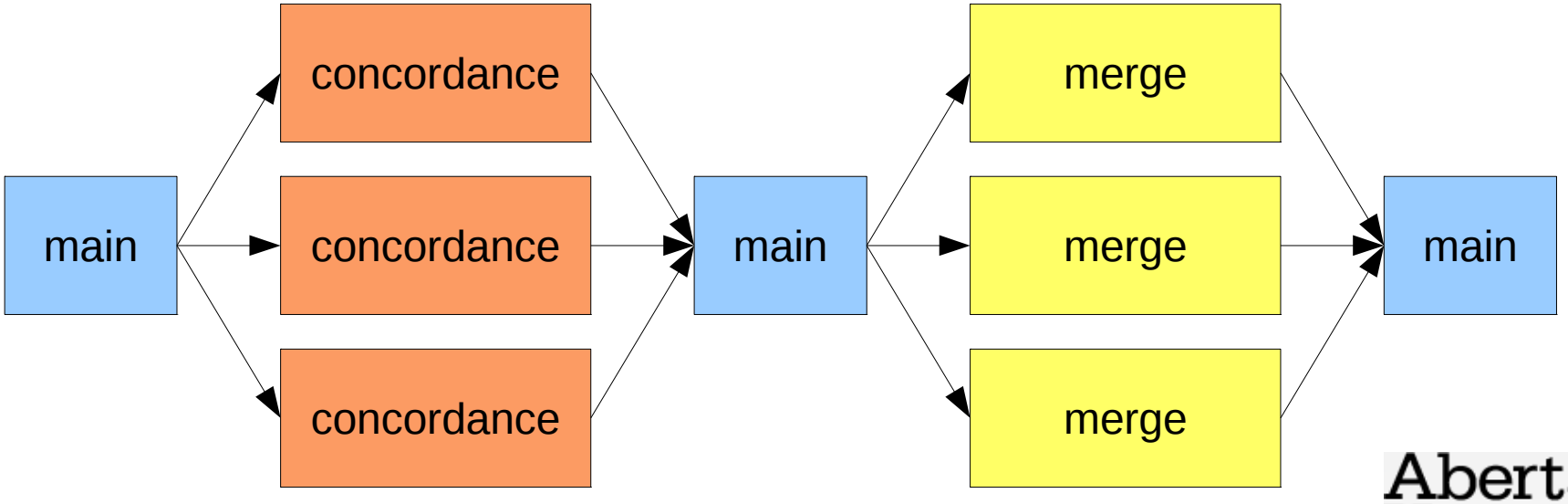
Version 3



Abertay University

Applying Rule 3

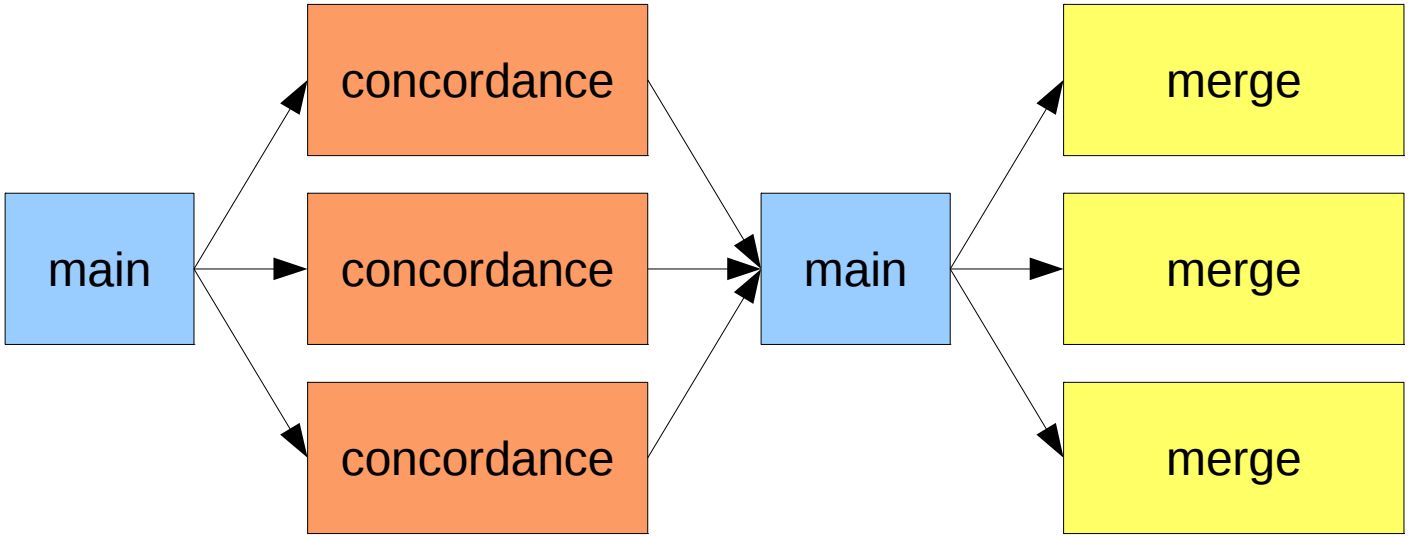
- That's even slower, although at least it scales...
- Break out the profiler: it's now spending most of its time communicating between processes
  - in **pickle**, Python's serialiser



Abertay University

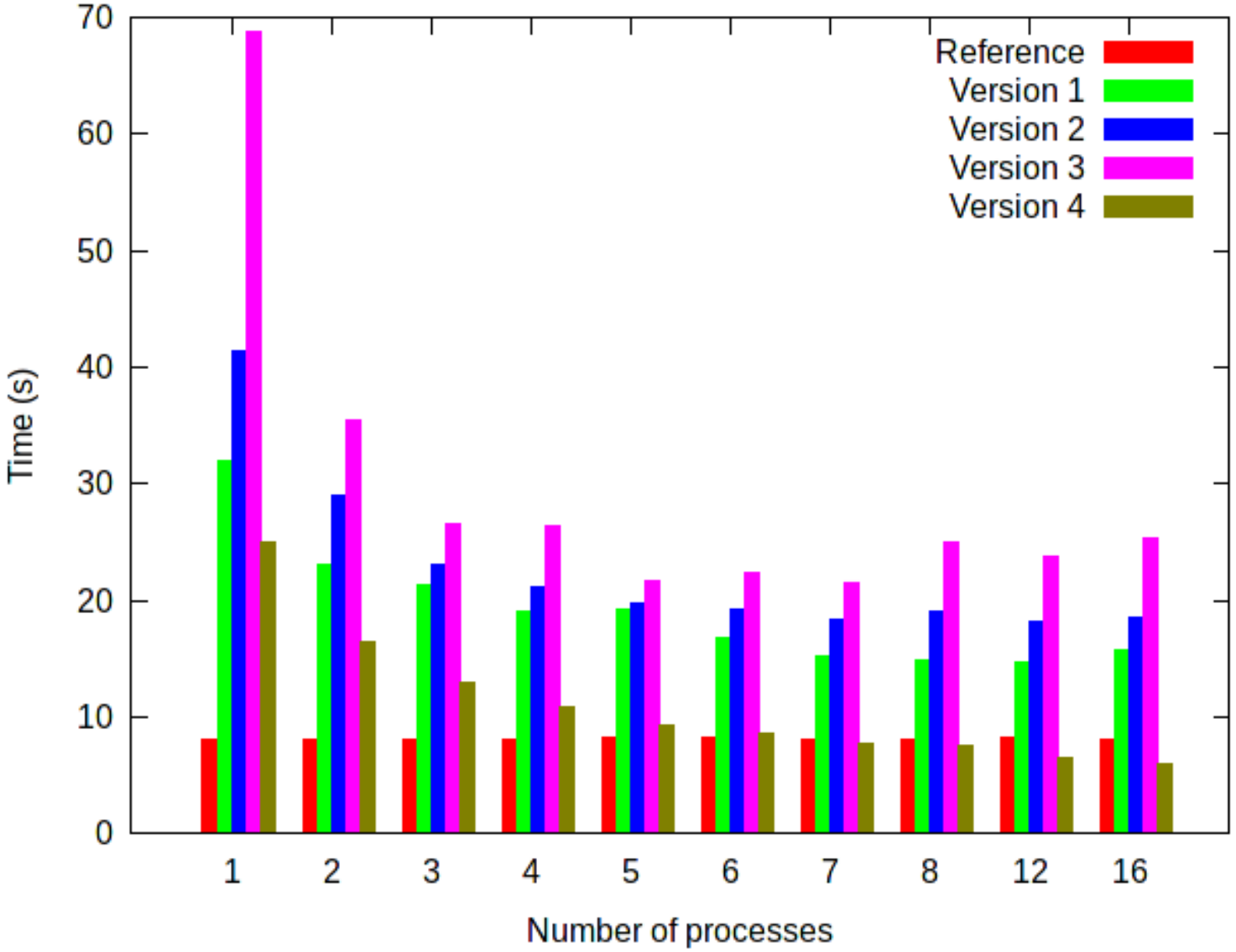
Arrow removal, stage 1

- Parallelise the output
- Each merge writes its own output, serialised using a **Lock**
- No less work to do – but less communication



Abertay University

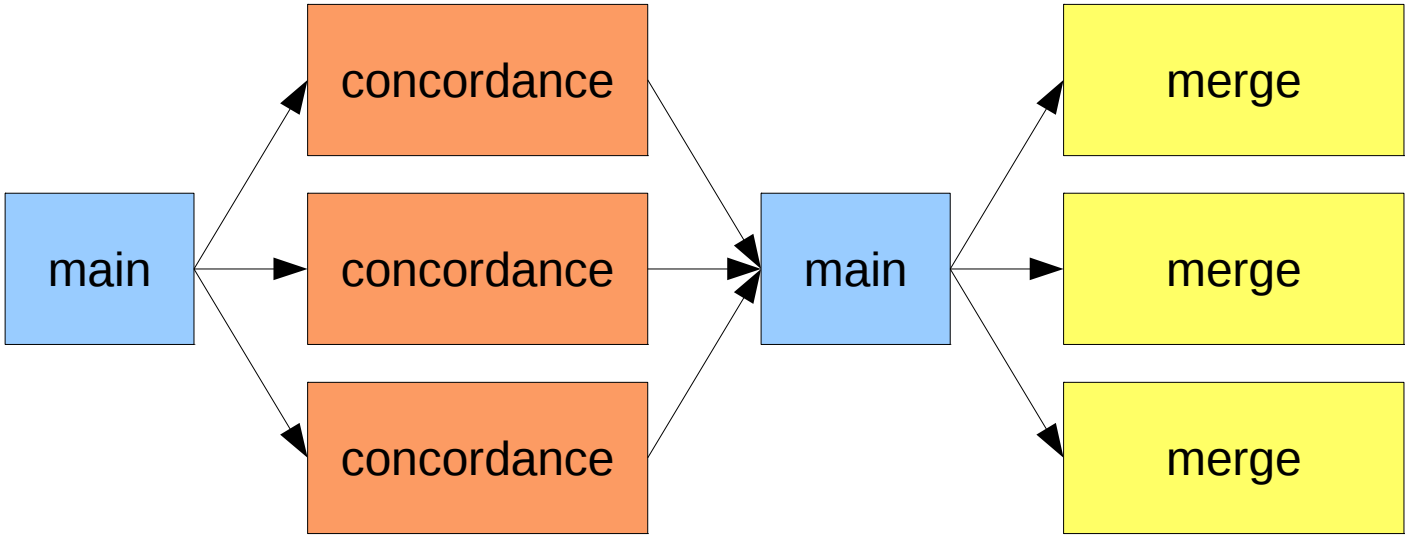
Version 4



Abertay University

Aha!

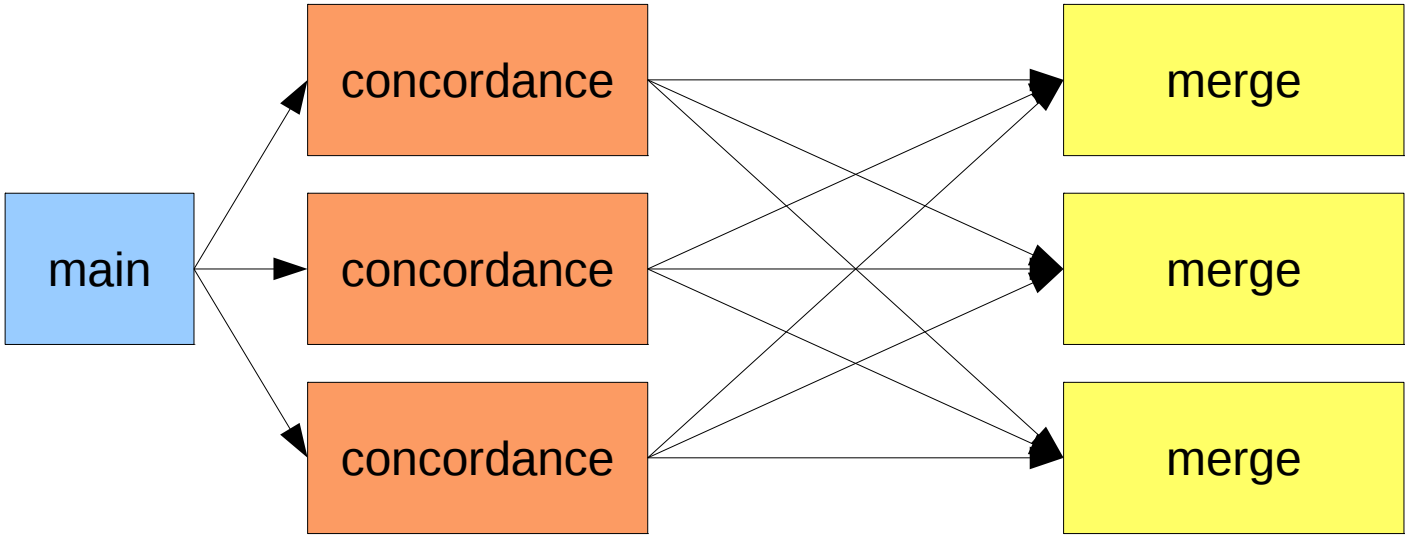
- We're beating the original version now!
- Let's keep going along those lines...



Abertay University

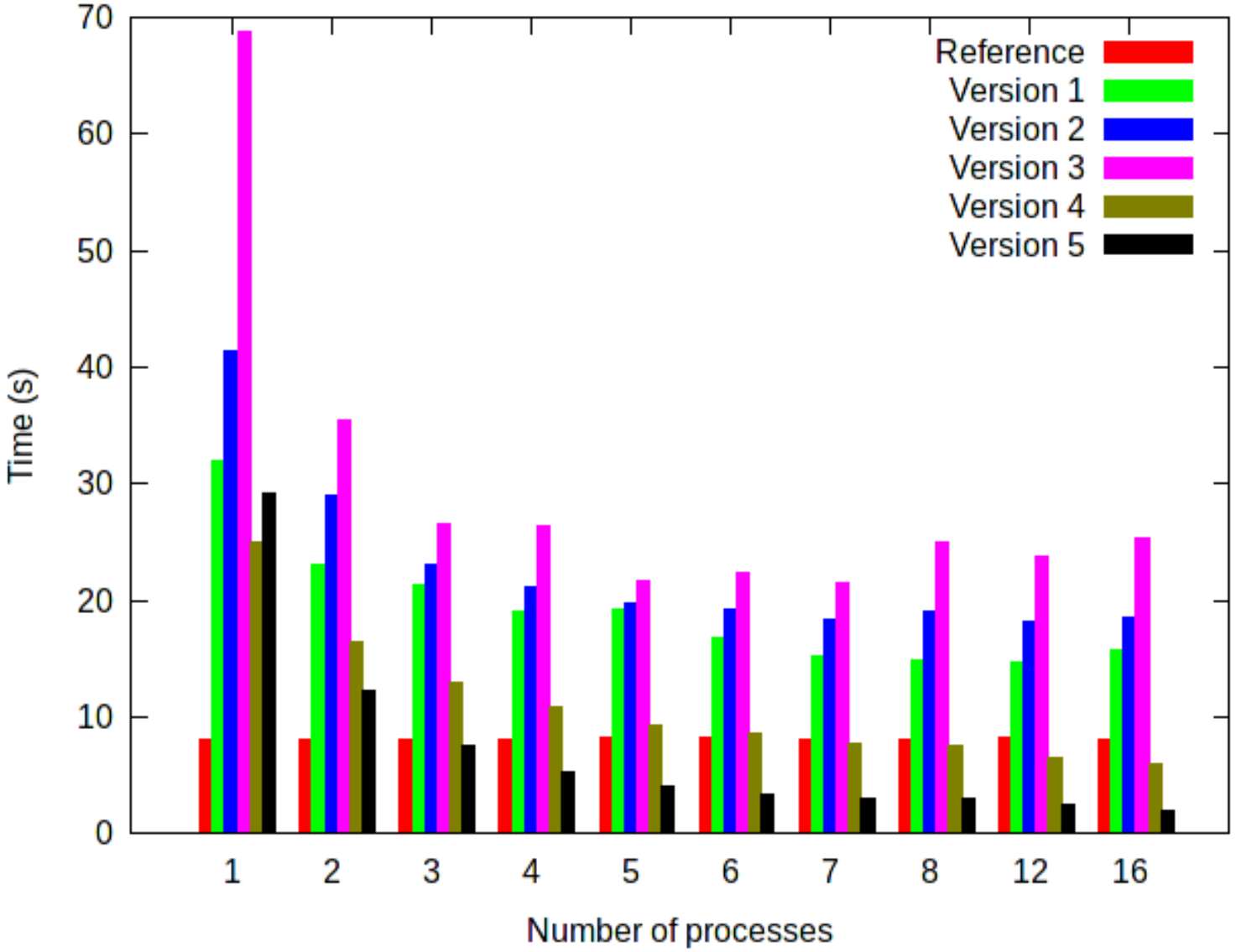
Arrow removal, stage 2

- Give each merge an incoming **Queue**
- Connect concordances directly to merges
- Each phrase only communicated once...
  - … and the communication is parallelised too



Abertay University

Version 5



Abertay University

Success

- We beat the original version at 3 cores, and it hasn't hit a bottleneck by 16
- Even better: it's scaling linearly!
  - Using **N** cores requires **1/N** time
- This is a **concurrent** solution – giving the kernel more freedom to schedule efficiently
  - … and how I would have built it in the first place using a process-oriented approach

Abertay University

Summing up

- “Do the simplest thing that can possibly work”
- Profile first
- All the improvement has come from changing the **structure** of the program
- **No shared memory** – this is a message-passing solution, amenable to distribution
- Could optimise the sequential bits – but this is probably fast enough now; CPUs are cheap...

Any questions?

- Thanks for listening!
- Get the code:
  git clone http://offog.org/git/sicsa-mcc.git
- Contact me or get this presentation:
  http://offog.org/

Abertay University