Melbourne Bioinformatics Lab Talk, 3 August 2018

Memory efficient parallelisation of HiTIME in C++

Bernie Pope, bjpope@unimelb.edu.au

Outline

- Twin ion mass spectrometry
- HiTIME algorithm and initial implementation in Python
- Memory efficient parallelisation in C++
- Performance
- Discussion

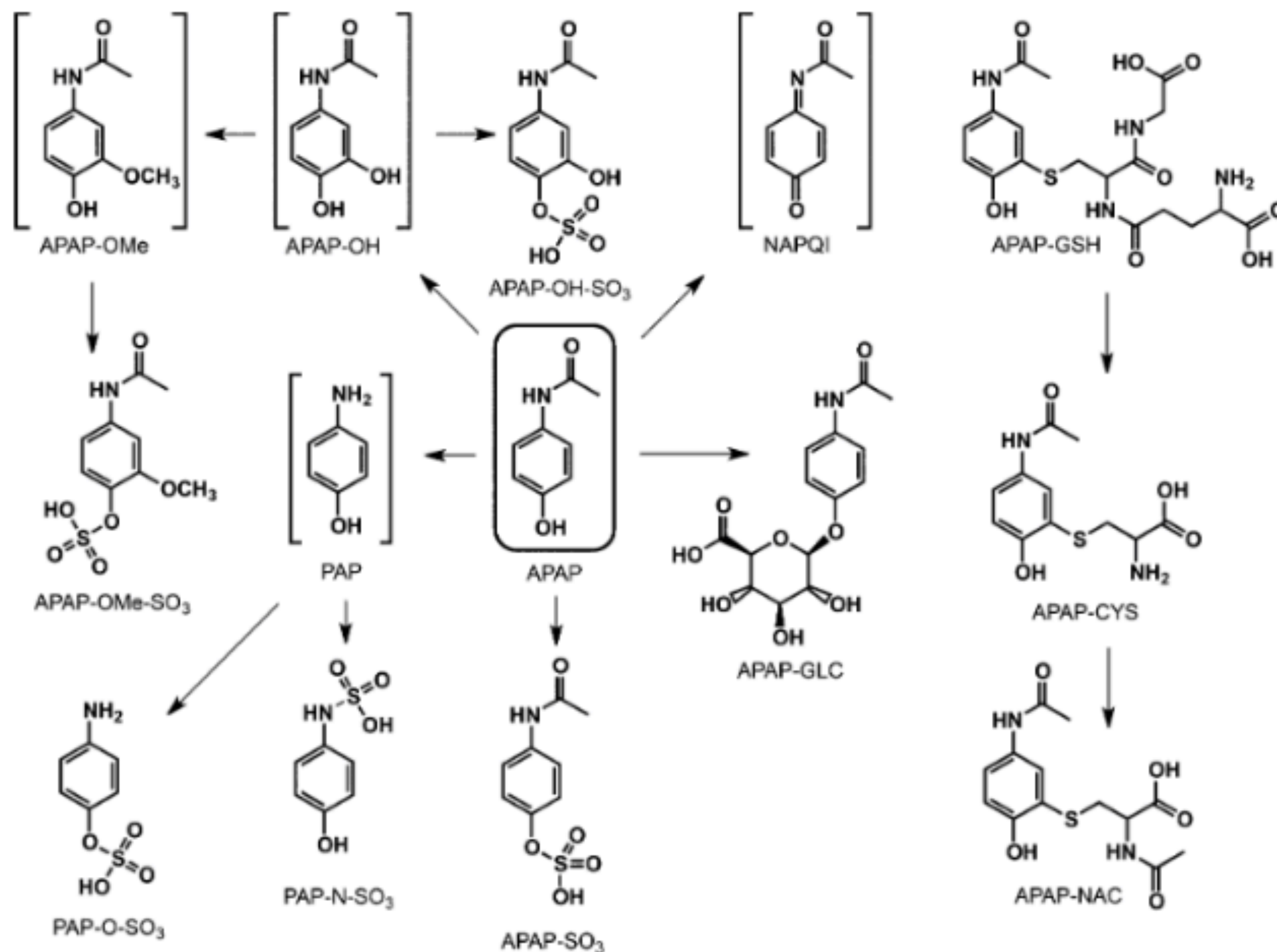
Twin Ion Mass Spectrometry

- Typical scenario:
    - A new drug is being tested.
    - As it is metabolised new compounds (called metabolites) are formed.
    - These metabolites can affect the performance of the drug and cause side effects.

Twin Ion Mass Spectrometry

Paracetamol (APAP) metabolites

observe that the benzene ring remains intact



Leeming et al. Analytical Chemistry, 2015

Twin Ion Mass Spectrometry

- We want to discover these (novel) metabolites, typically from a blood sample.
- We could use mass spectrometry to identify all compounds in the blood, but …
- … blood contains thousands of other molecules which could be confused with true drug metabolites.
- Proverbial needle in a haystack problem.

Twin Ion Mass Spectrometry

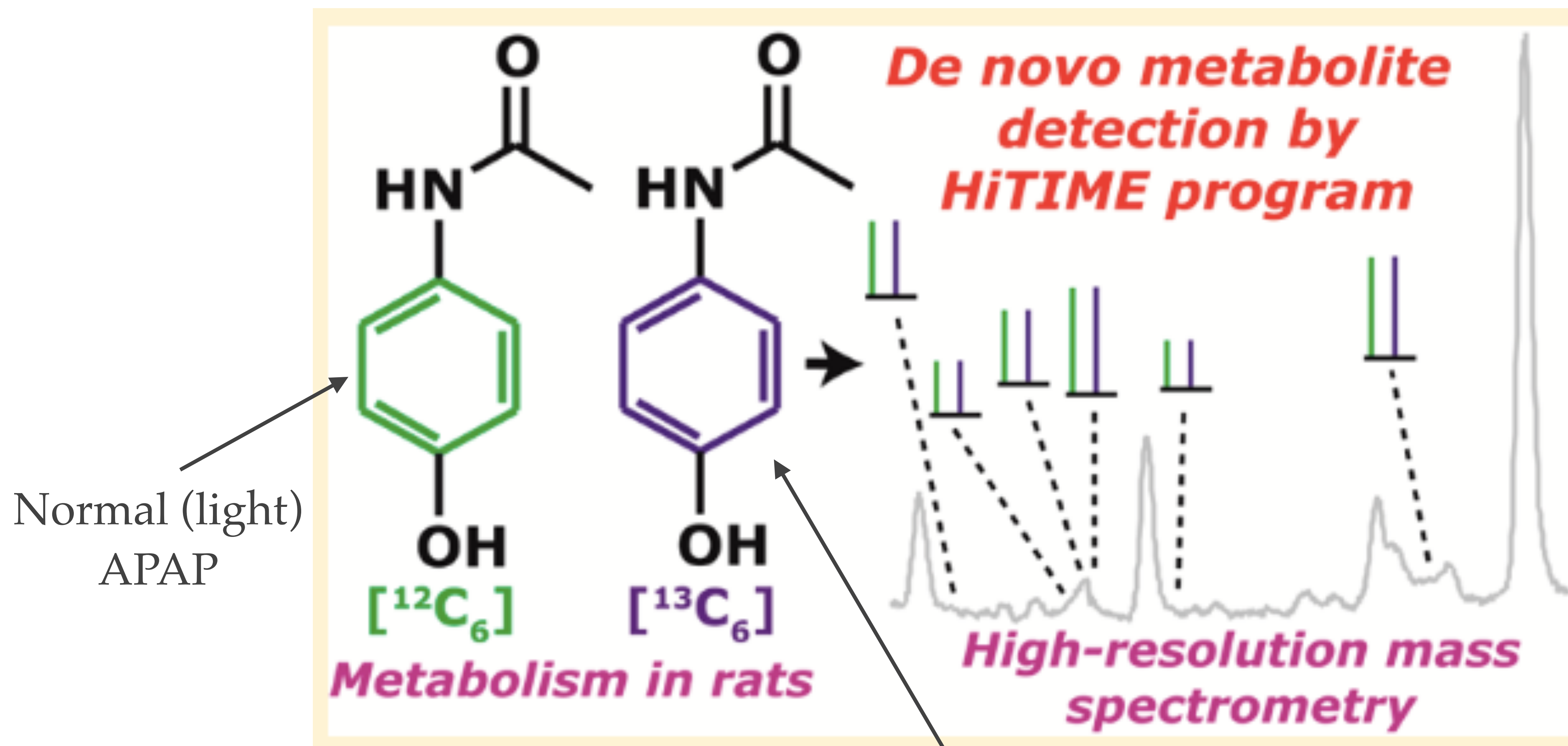
- Solution:
  - Make two versions of the drug molecule:
    1. normal
    2. heavy
  - In APAP, a heavy version can be made by substituting ^{12}C with ^{13}C in the benzene ring.
  - Observe that the normal and heavy versions have an expected mass difference (about 6 Da).
  - Despite the mass difference the two versions ought to have the same chemical properties.

Twin Ion Mass Spectrometry

- Solution continued:
  - Administer a 1:1 mixture of normal and heavy versions of the drug to the test organism.
  - Allow time to metabolise, then take a blood sample and run it through a mass spectrometer.
  - Search for pairs of peaks (twins) in the resulting spectra which:
    - have the expected mass difference
    - equal intensity (abundance)
    - appear at the same time
  - This twin ion signal is unlikely to occur by chance.

Twin Ion Mass Spectrometry

Example of Paracetamol (APAP)

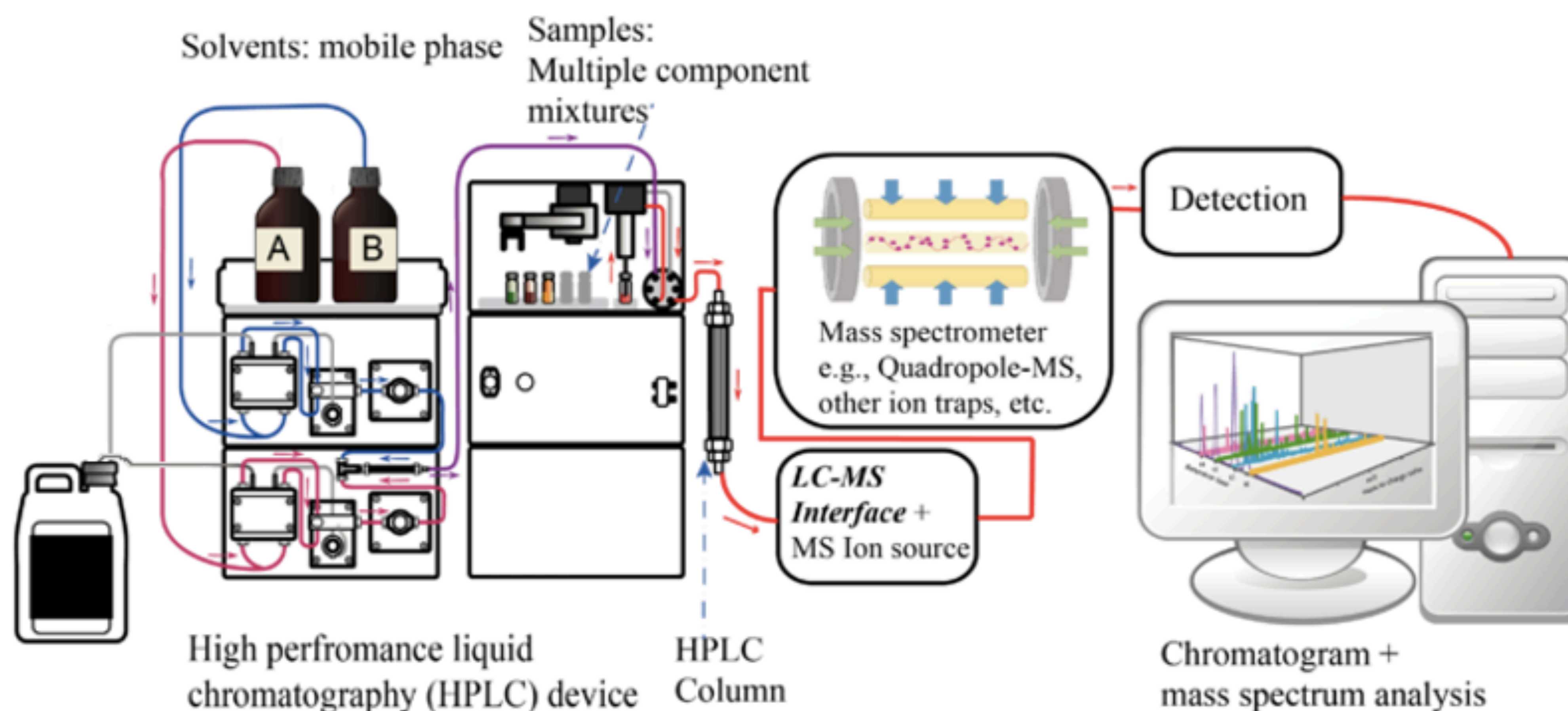


Heavy APAP = +6.0201 Da

Leeming et al. Analytical Chemistry, 2015

Twin Ion Mass Spectrometry

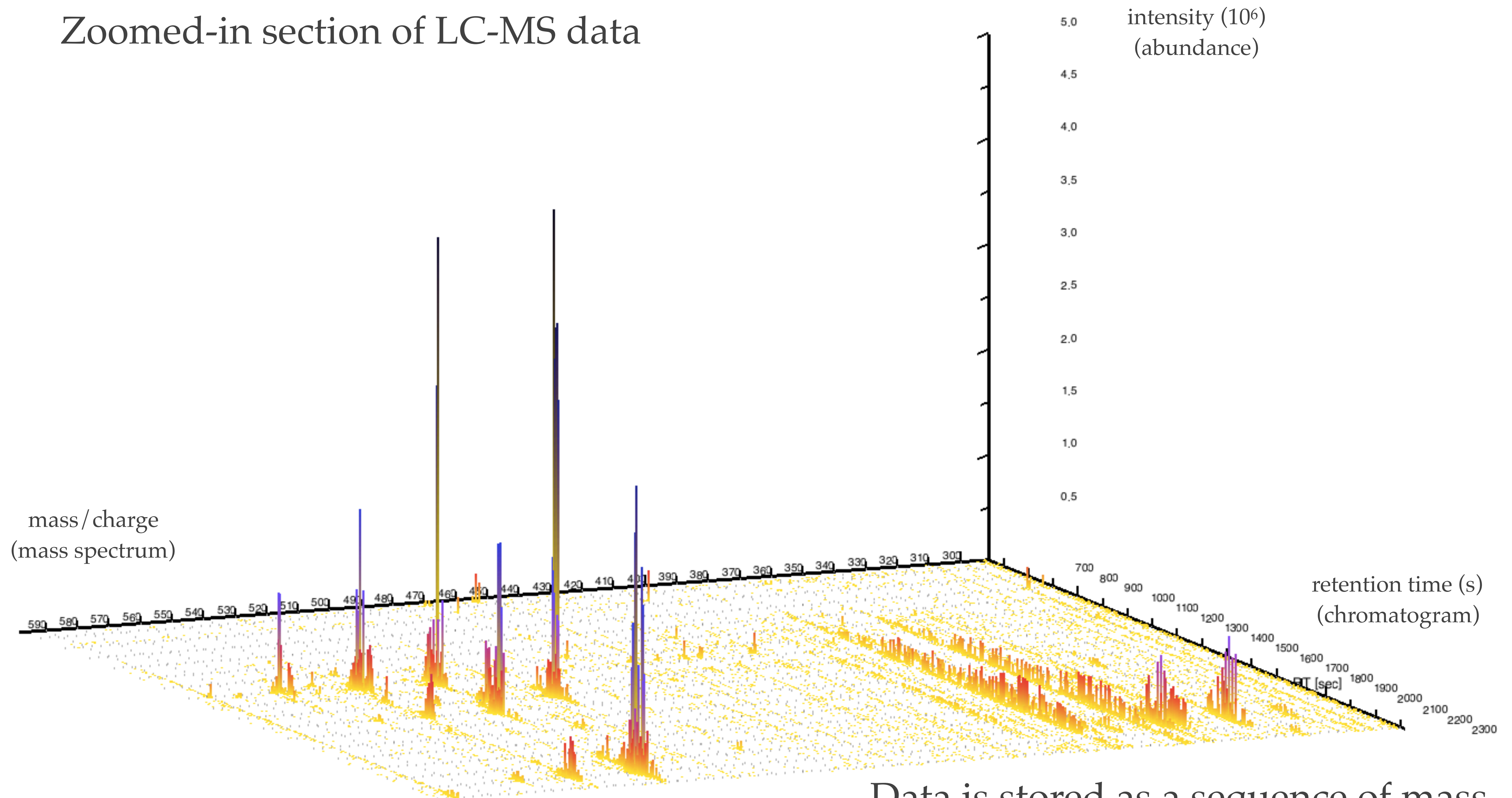
Liquid Chromatography - Mass Spectrometry:
compounds in sample are separated by chromatography,
before being sent to mass spectrometer. This adds a time dimension.



https://commons.wikimedia.org/wiki/File:Liquid_Chromatography_Mass_Spectrometer.png

Twin Ion Mass Spectrometry

Zoomed-in section of LC-MS data



Data is stored as a sequence of mass spectra in retention time order

HiTIME algorithm

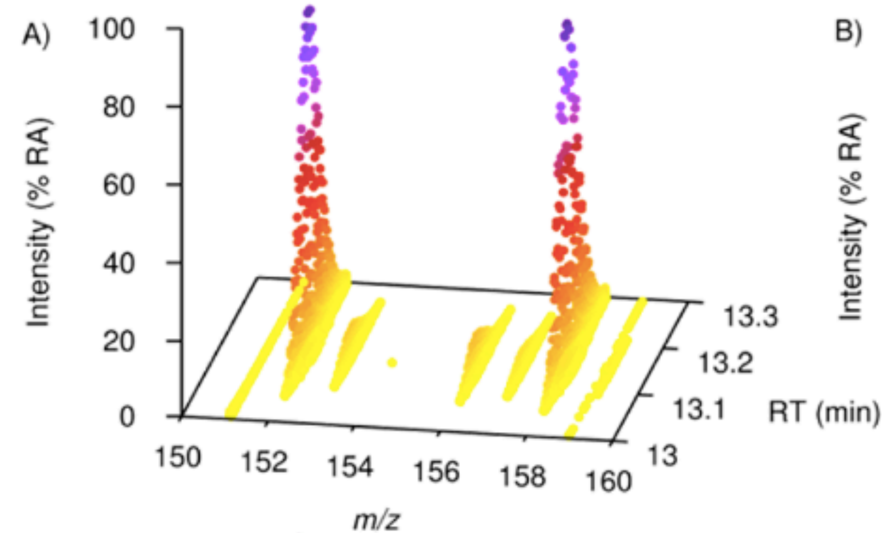
- High resolution twin-ion metabolite extraction.
- An algorithm for the detection of twin-ion signals in LC-MS data.
- Joint work with Andrew Isaac, Michael Leeming, Richard O'Hair, William Donald and others.
- Published in Analytical Chemistry, 2015.

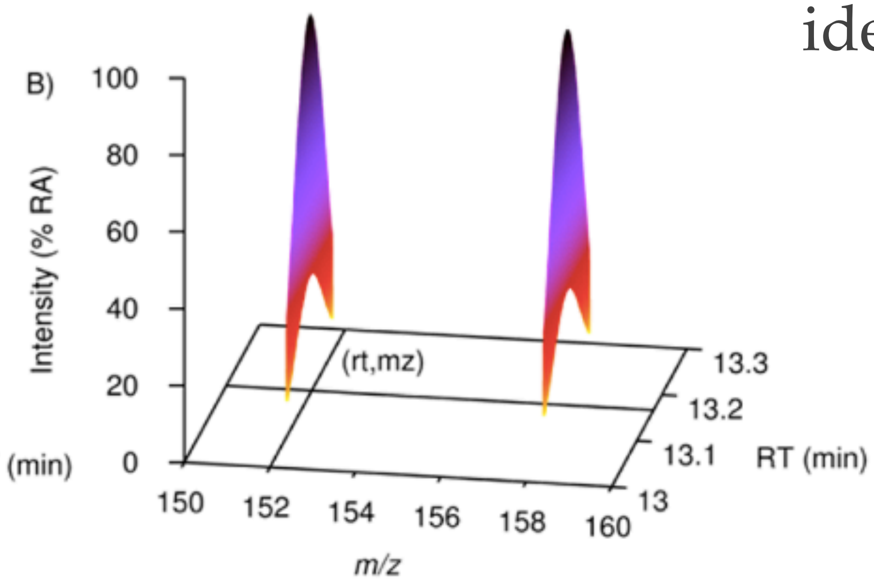
HiTIME algorithm

- Stencil convolution:
  - Each data point in the input is scored based on some function over its local neighbourhood of data points.
  - Can be implemented as a sliding window.
  - An example is the Sobel differentiation operator used in image edge detection.
- HiTIME is a kind of stencil convolution, but the computation at each point considers two windows. One for the normal mass, and one for the heavy mass.

HiTIME algorithm

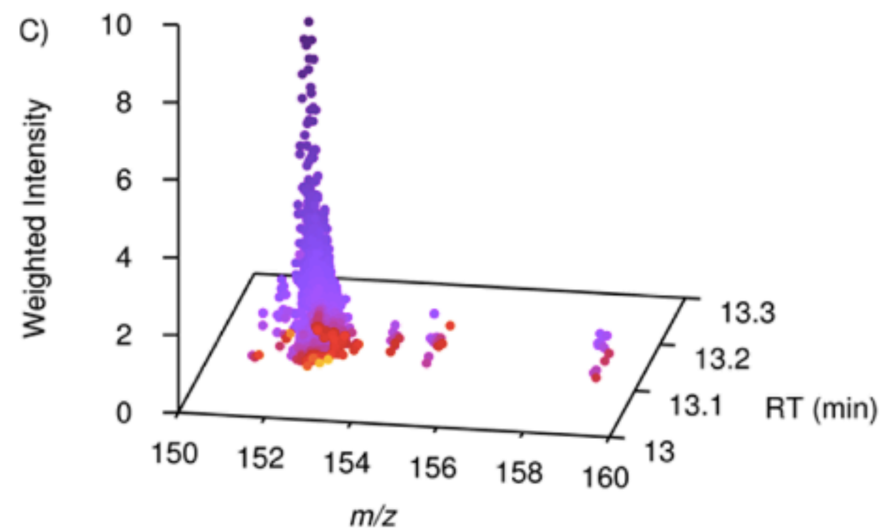
input data showing twin ion peaks

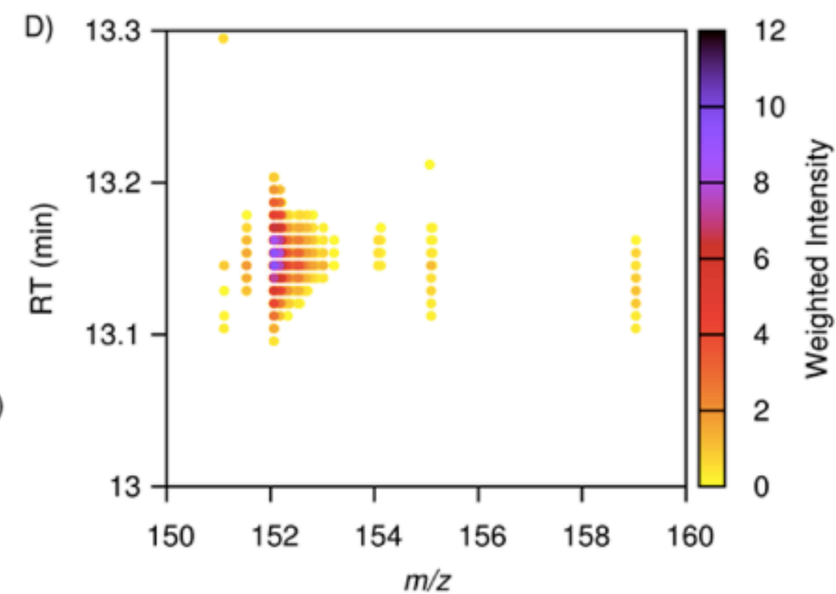




ideal twin ion model

output scored data centred on the normal mass isotope





top down view of output as heat map

HiTIME algorithm

```
output_spectra = []
for spectrum in time_sorted(input_spectra):
    output_spectrum = []
    for (mass, intensity) in mass_sorted(spectrum):
        low = get_neighbourhood(mass)
        high = get_neighbourhood(mass + delta)
        new_intensity = score(intensity, low, high)
        output_spectrum.append(mass, new_intensity)
    output_spectra.append(output_spectrum)
```

HiTIME algorithm

```
output_spectra = []
for spectrum in time_sorte
    output_spectrum = []
    for (mass, intensity) in mas          um):
        low = get_neighbourhood(mass)
        high = get_neighbourhood(mass + delta)
        new_intensity = score(intensity, low, high)
        output_spectrum.append(mass, new_intensity)
    output_spectra.append(output_spectrum)
```

Requires access to neighbouring spectra +/- in time

HiTIME algorithm

- The original implementation of HiTIME was in Python.
- This enabled rapid prototyping.
- However, runtime performance was not good.
- We parallelised the code using MPI, and used 480 cores to complete one large input sample in under an hour.
- Not convenient and users don't always have ready access to HPC clusters.

HiTIME algorithm

- Our goal was to reimplement the algorithm to be time and space efficient and thus be usable on modest hardware.
- We chose C++ for raw sequential performance and because of good mass spectrometry support from the OpenMS library (www.openms.de).
- Original sequential port to C++ was done by VLSCI intern Luke Zappia using the ProteoWizard library.

A memory inefficient parallelisation

- Stencil convolutions are popular in HPC because they are data parallel, therefore relatively easy to parallelise <u>if you have all your input data in memory</u>.

A memory inefficient parallelisation

- Load all the input spectra into memory from input file.
- Partition the spectra into equal sized time segments. One for each CPU thread.
- Apply the HiTIME algorithm to each segment in parallel.
- Write output spectra to memory (necessary because of parallel execution over segments).
- When all CPU threads are finished, write output spectra to file in time order.

A memory inefficient parallelisation

- This parallelisation is easy to implement and scales well, but requires at least 2N memory, where N is the size of the input data.
- We have high resolution data sets which are 14GB in size.
- Ideally we want an algorithm which does not need to keep all spectra in memory at once, and can write output spectra to file as the computation proceeds.

Memory efficient parallelisation

- Leapfrog. Each thread works on the next unsolved spectrum, in retention time order:

```
next_spectrum_id = get_next_spectrum_todo()
while (next_spectrum_id < num_spectra):
    output_spectrum = score(next_spectrum_id)
    put_spectrum(next_spectrum_id, output_spectrum)
    next_spectrum_id = get_next_spectrum_todo()
```

Memory efficient parallelisation

- Challenges:

  1. Threads are working on highly overlapping windows of spectrums. We don't want to re-read the same spectrum from file many times (nor have multiple copies in memory).

  2. We cannot guarantee that threads will complete in time order, so we may not be able to immediately write an output spectrum to file. Some in-memory reordering may be needed.

Memory efficient parallelisation

- Solution to challenge 1:
    - Keep a cache of spectra that are currently being used. Share this between threads.
    - If the cache is efficient, we should only ever read each spectrum once from file and keep only one copy in memory.
    - What cache retention policy?
    - We chose a least recently used (LRU) cache because it is fast to implement and likely to give good temporal locality.
    - OpenMS provides a IndexedMzMLFileLoader, which allows random access to spectrums within the input file.

Memory efficient parallelisation

- Challenge 1b:
  - But what if we evict a spectrum from the LRU cache but it is actually needed by one or more executing threads (i.e. cache is too small)?
  - We must keep it in memory even though it isn't in the cache. We must free its memory when no longer needed by running threads.

Memory efficient parallelisation

- Solution to challenge 1b:
    - The input spectrum LRU cache holds C++ shared pointers (`shared_ptr`) to spectrums.
    - Shared pointers are reference counted. So they hold onto the data until the last remaining reference is deallocated.

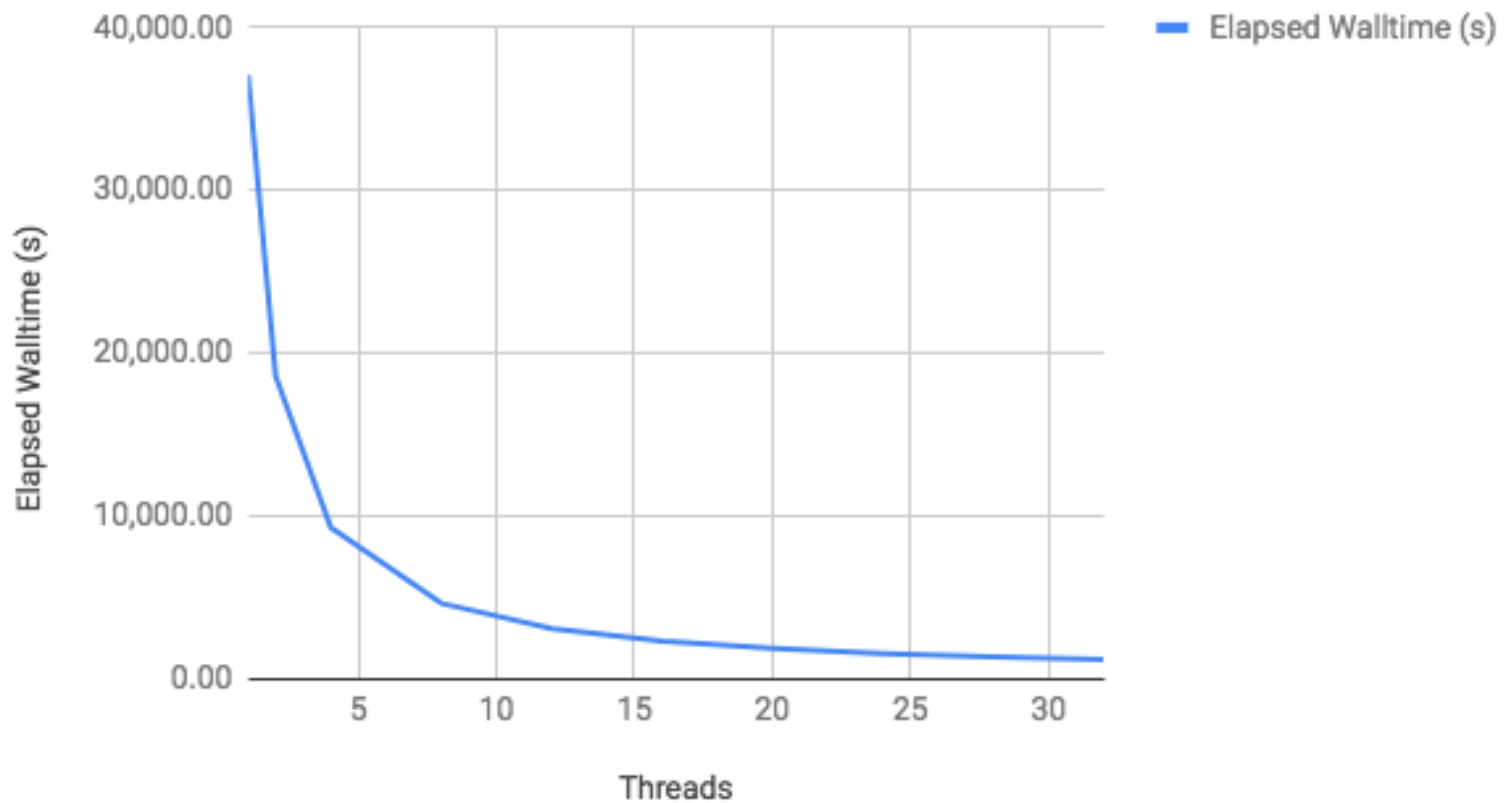
Memory efficient parallelisation

- Solution to challenge 2:
    - Keep a priority queue (heap) of output spectra, ordered by (retention) time.
    - A spectrum is only written to disk when it is the next one pending in time order.
    - If each thread does roughly the same amount of work then the queue will not grow very large.
    - Leapfrog means that we generally work on input spectra in retention time order.

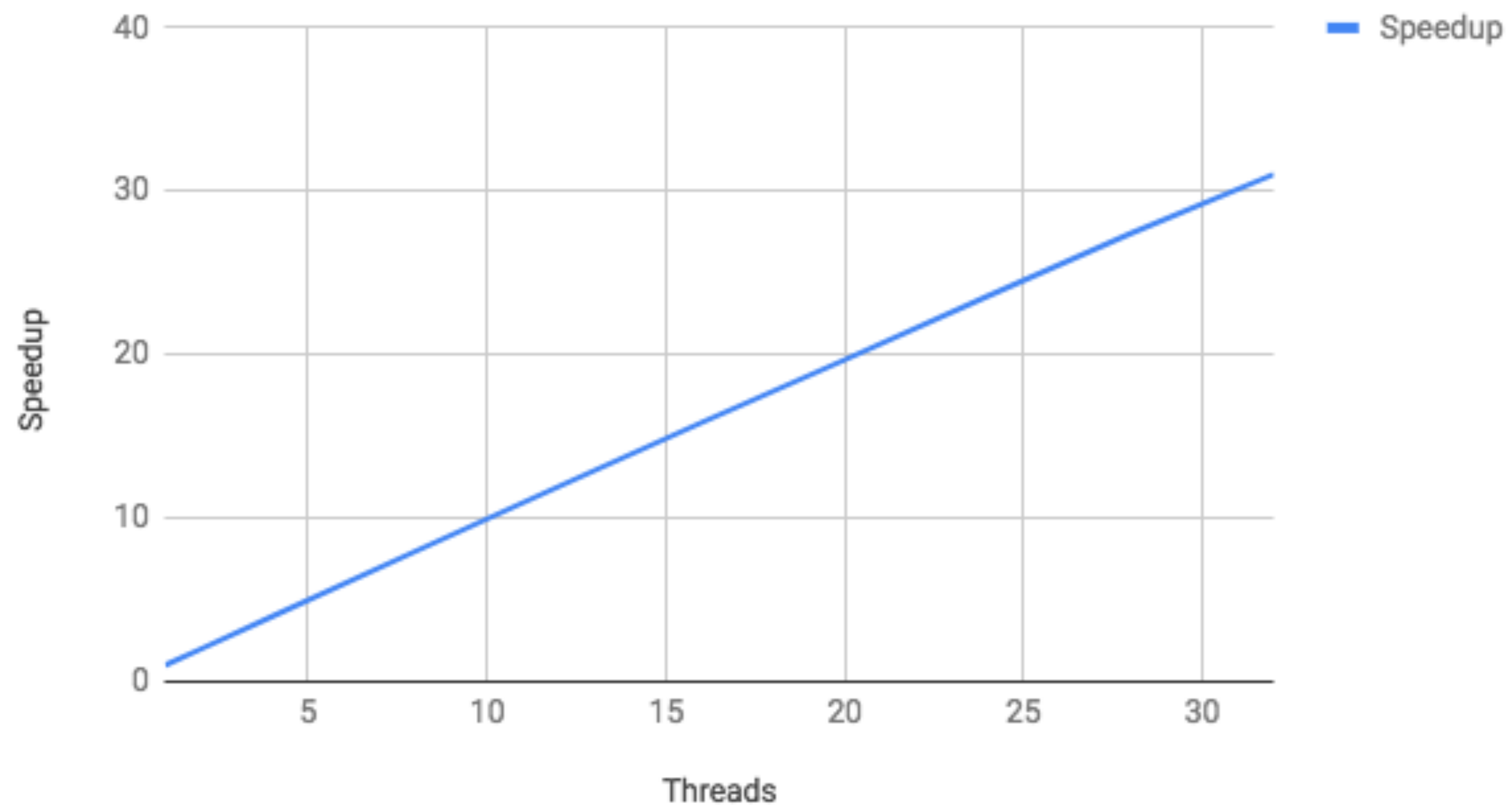
Performance

- Input test data is a 14 GB high resolution LC-MS data file from a twin-ion experiment for APAP metabolites. Same data as previous publication.
- Recorded elapsed wall-time and maximum resident set size (RSS) for HiTIME-CPP on 1 to 32 CPU cores on Snowy (2.3GHz Intel Xeon E5-2698 v3, 32 cores per node).

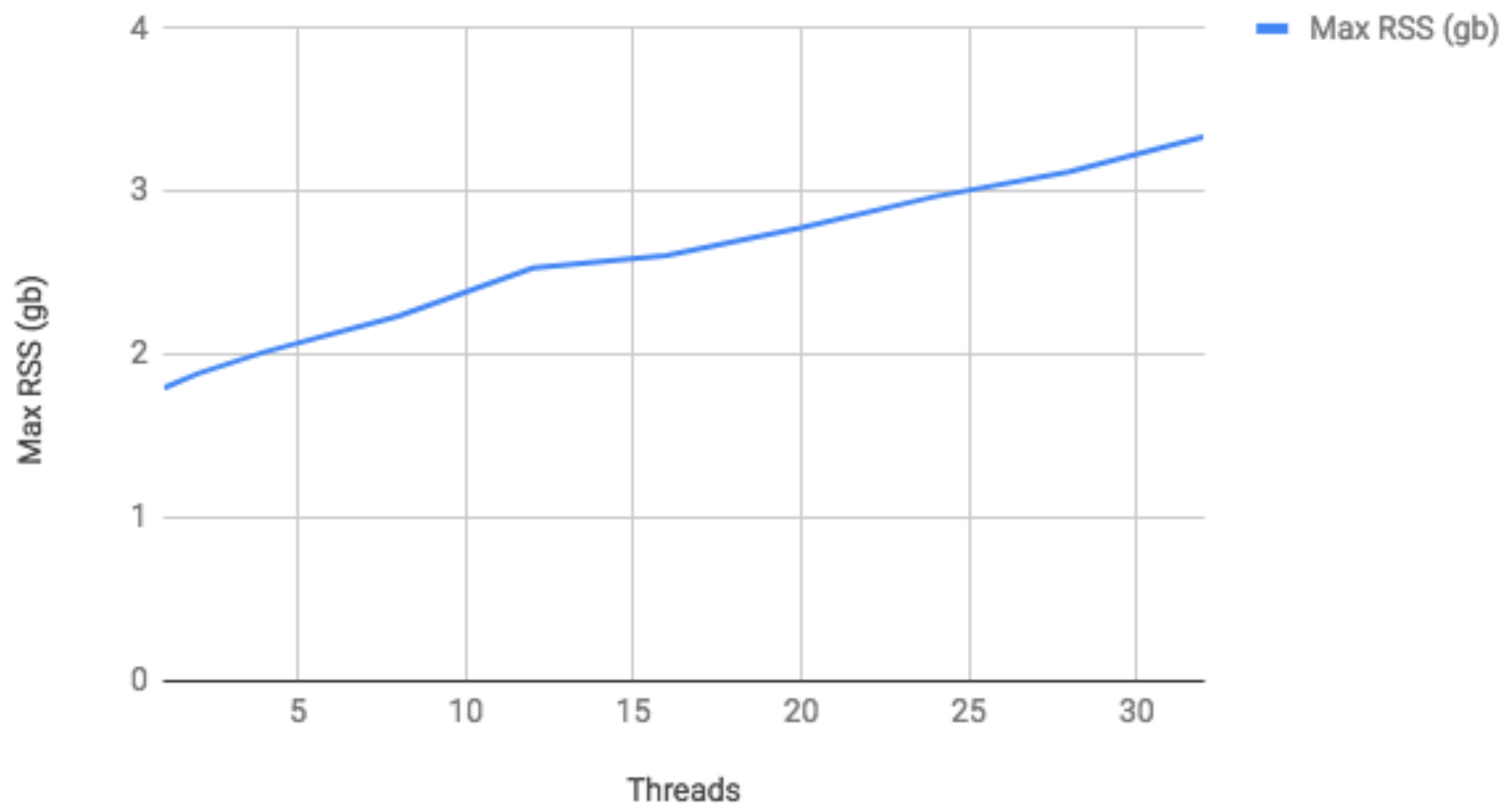
Performance



Performance



Performance



Discussion

- All parallelisations produce bit-identical output. But we still have some correctness testing to do.
- We also made memory and sequential time performance improvements on the HiTIME implementation along the way, which were made somewhat easier by the natural code refactoring that happened during the parallelisation.
- Modern C++ has lots of nice features, including easy-to-use threading, however library packaging is far behind other languages.
- Building the code is challenging due to OpenMS dependencies and its use of CMake.
- We made a Docker container to make this easier and reproducible. Has nice benefits for Travis CI too.
- Andrew Isaac will discuss the inner workings of the scoring algorithm in an upcoming lab talk.
- Code is here: https://github.com/bjpop/HiTIME-CPP