Parsec- A ‘Big Data’ Component Detection Algorithm and Framework

Iman Mohtashemi, Janos Fodor Kis, Mathew Kump, Vijay Kulkarni, Thermo Fisher Scientific, 355 River Oaks pkwy, San Jose, Ca, USA, 95134

ABSTRACT

Purpose: Demonstration of a scalable component detection algorithm with near constant time complexity when scaled across compute clusters

Methods: A map-reduce implementation of component detection using the Apache Spark™ cluster-computing framework is demonstrated using Mobius (C#) and Python implementations.

Results: We demonstrate a component detection algorithm that leverages the distributed compute framework (Apache Spark) which results in several orders of magnitude of performance. We further demonstrate a near constant time complexity on 1000 serum metabolomics raw files providing a framework to build large scale statistical power in peak picking/component detection studies.

INTRODUCTION

Component detection is a computationally intensive process of data reduction of highly redundant LC-MS data to biologically meaningful compound results. To our knowledge, no component detection algorithm exists that can process and scale on massive LCMS data sets. ParSeC (Parallel Sequence Component Detection) solves this problem by implementing a component detection workflow using the map reduce programming model. The algorithm is naturally distributed across any cluster. The algorithm is simple and scalable. Most importantly, all the delegate functions (e.g., peak detection, isotope clustering or component assembly), are completely interchangeable. In a compute cluster environment, all steps of the workflow can be parallelized. Initial prototype proof of concept (POC) results show speed improvements of several orders of magnitude with near constant time scalability. The algorithm / framework described will ultimately enable users to analyze massive data sets not previously possible, build statistical power in their studies and scale dynamically and on-demand.

MATERIALS AND METHODS

We demonstrate proof of concept in two phases.

- An existing legacy algorithm is modified to be independent of Thermo Scientific™ .raw file format. Thermo Scientific LCMS .raw files are converted to the Parquet file format which is a free and open-source column-oriented data store of the Apache Hadoop™ ecosystem. We leverage the map-reduce programming model while gaining the 100x speed improvements of in-memory computing of Apache Spark vs. the traditional file-based map-reduce. Functions are modularized to be deployed at scale on any compute cluster (e.g. AWS, Azure, Google).
- A second python implementation of a more granulized and fully parallelizable Component Detection algorithm is also presented.
- Three High Resolution Accurate Mass (HRAM) data sets were evaluated:
    - 2 tea metabolomics LCMS runs ~ 40 Mb (qualitative analysis)
    - 109 beer metabolomics LCMS runs ~21 Gb (medium scale)
    - 1000 serum metabolomics LCMS runs ~ 112 Gb (large scale)

Slower approach, really only parallel processing raw file representations

```
var components = scanData
            .Map(KVCeMassTraces)
            .Map(KVCePeakDetect)
            .Map(KVCeIsotopePeaks)
            .Map(KVCeAssembleComponents);
return components;
```

Faster distributed approach distributing data at the scan level

```
features = spark.read.parquet(i) \
    .rdd.map(lambda scan: pc.kv_scan(scan, 1)).groupByKey() \
    .flatMap(lambda file: pc.bin_masses(file)) \
    .flatMap(lambda fileMassBin: pc.create_mass_traces(fileMassBin)) \
    .flatMap(lambda trace: pc.detect_peaks_peakutils(trace[1], sn)) \
    .map(lambda p: pc.map_peak(p,rtTolerance)) \
    .reduceByKey(lambda p1, p2: p1 + p2) \
    .map(lambda p: pc.DetectSmallMoleculeFeaturesFaster(p,err,False)) \
```

Component Detection

The primary goal of LCMS is to identify all the chemical components in a sample. However, LCMS data is often rich, redundant and requires several layers of data reduction to funnel toward a biologically relevant compound list. Many algorithms have been developed over the years termed ‘Component Detectors’ to do such data reduction. While there are many strategies employed (e.g. the choice of peak detection models, isotope clustering and ordering of steps) they all follow a general funneling pipeline. The input data is a mass spectrum in the time domain and as such spectra are clustered, grouped, peak detected, charge state and adduct correlated and reduced to a compound list. Data reduction can be several orders of magnitude. Many of these individual steps can be computationally intensive and such workflows are prohibitive in large data sets. We demonstrate a map-reduce data reduction strategy as described below.

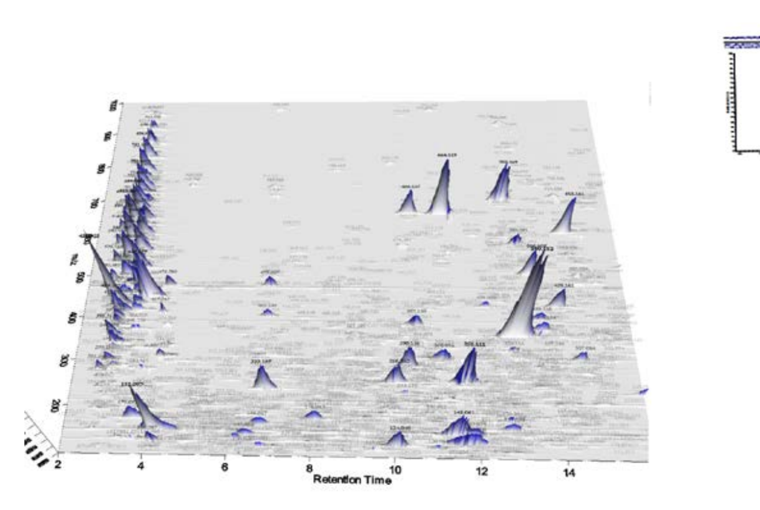


Figure 2. LCMS Feature Map

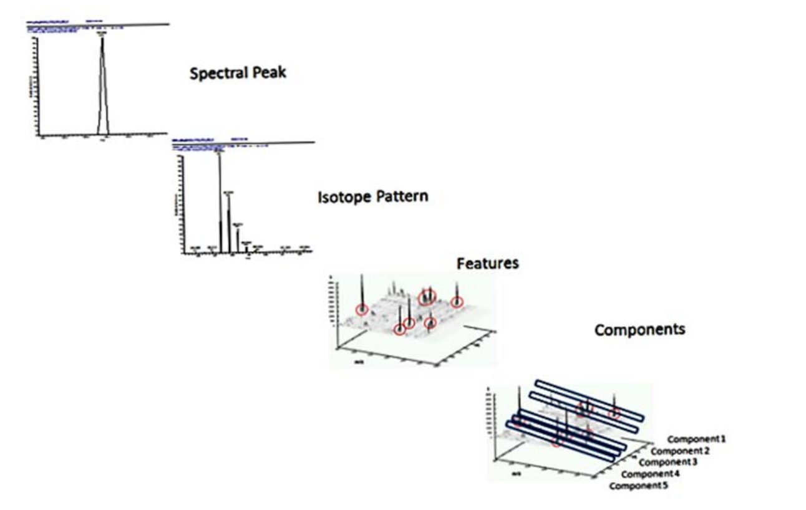


Figure 3. Component Detection data reduction pipeline

Motivation (Map-Reduce)

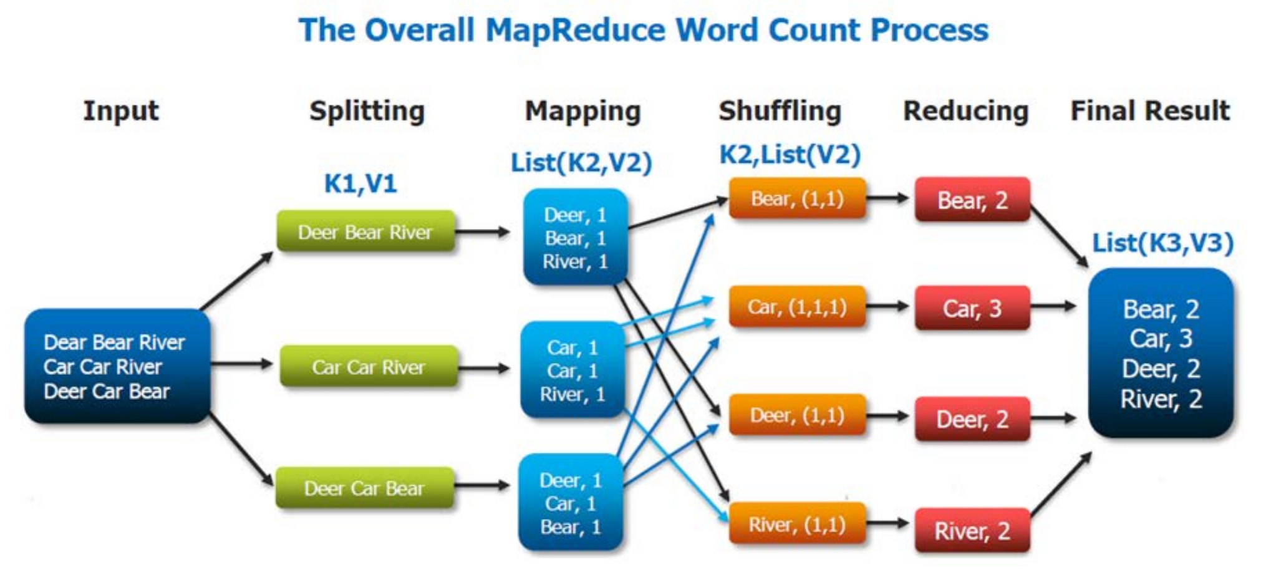


Figure 4. Canonical map-reduce example. Words are counted from an arbitrary number of files

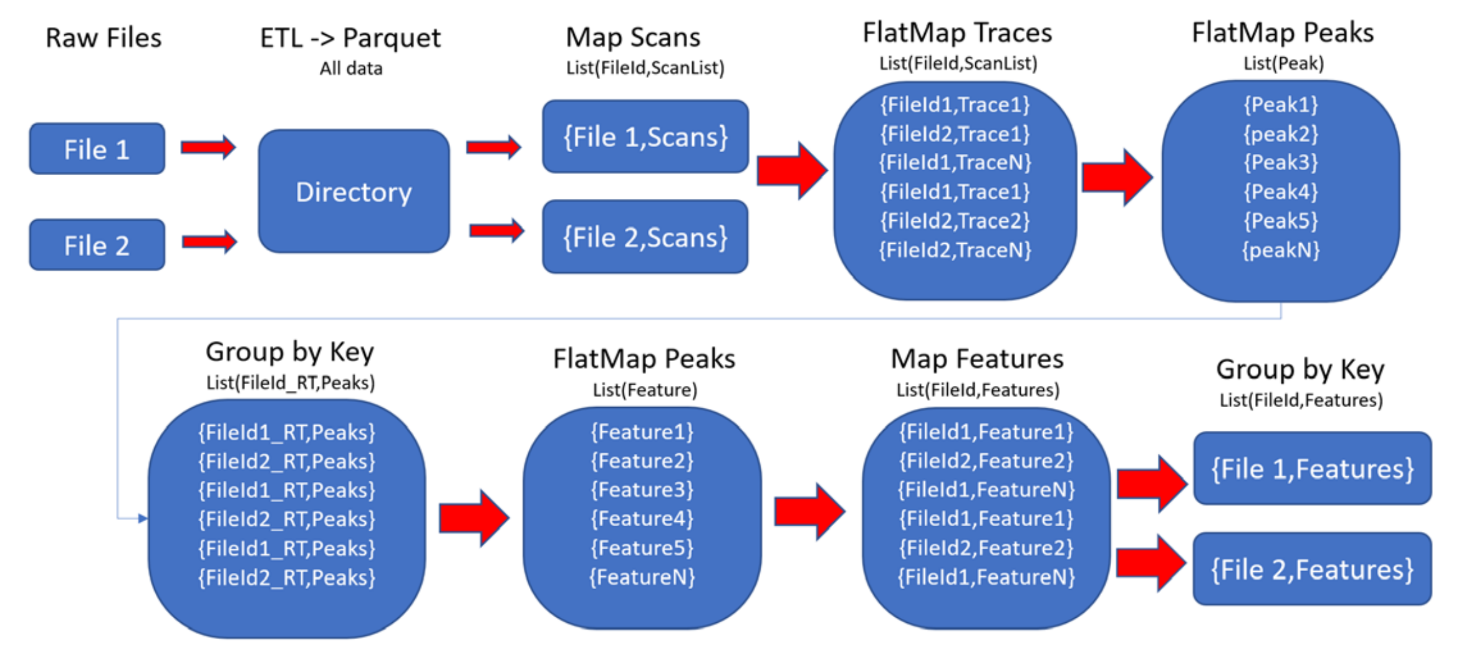

Raw Files
File 1
File 2
ETL -> Parquet
All data
Directory
Map Scans
List(FileId,ScanList)
{File 1,Scans}
{File 2,Scans}
FlatMap Traces
List(FileId,ScanList)
{FileId1,Trace1} {FileId2,Trace1} {FileId1,TraceN} {FileId1,Trace1} {FileId2,Trace2} {FileId2,TraceN}
FlatMap Peaks
List(Peak)
{Peak1} {peak2} {Peak3} {Peak4} {Peak5} {peakN}
Group by Key
List(FileId_RT,Peaks)
{FileId1_RT,Peaks} {FileId2_RT,Peaks} {FileId1_RT,Peaks} {FileId2_RT,Peaks} {FileId1_RT,Peaks} {FileId2_RT,Peaks}
FlatMap Peaks
List(Feature)
{Feature1} {Feature2} {Feature3} {Feature4} {Feature5} {FeatureN}
Map Features
List(FileId,Features)
{FileId1,Feature1} {FileId2,Feature2} {FileId1,FeatureN} {FileId1,Feature1} {FileId2,Feature2} {FileId1,FeatureN}
Group by Key
List(FileId,Features)
{File 1,Features}
{File 2,Features}

Figure Map-reduce implementation of Component Detection (Only Feature Detection Shown)

A hands on example

https://notebooks.azure.com/im281/libraries/parsecdemo

jupyter Notebooks hosted on Microsoft Azure

Parallel injestion of Parquet files

Parquet is a distributed file format. The suttle but important part of this workflow is that we do not directly work with raw files. We work with a 'representation' of the raw file which can simply be characterized as a collection of scans. This is the signal we are concenrned with and all further processing will leverage this concept. As such we have converted the files to the parquet format. Parquet is a distributed file format. It has several advantages including columnar storage: fast column queries that benefits reading only few columns for all rows (all scans in all raw files) and is a compressed format. In Apache Spark, there is support for parallel injestion of all files using a simple read command. We simply point to a directory and read in all the files as distributed objects.

```
In [5]: #read all parquet files in a directory. Note that collect() is an action sending all the data to the driver
        scans = spark.read.parquet('data/*').take(5)
        for s in scans:
            print(s.FileName, s.RetentionTime)

        blacktea_1.raw 6.180753333333335
        blacktea_1.raw 6.211921666666667
        blacktea_1.raw 6.243086666666667
        blacktea_1.raw 6.273919999999995
        blacktea_1.raw 6.303753333333333
```

Mapping Raw Files * Optimization Opportunity

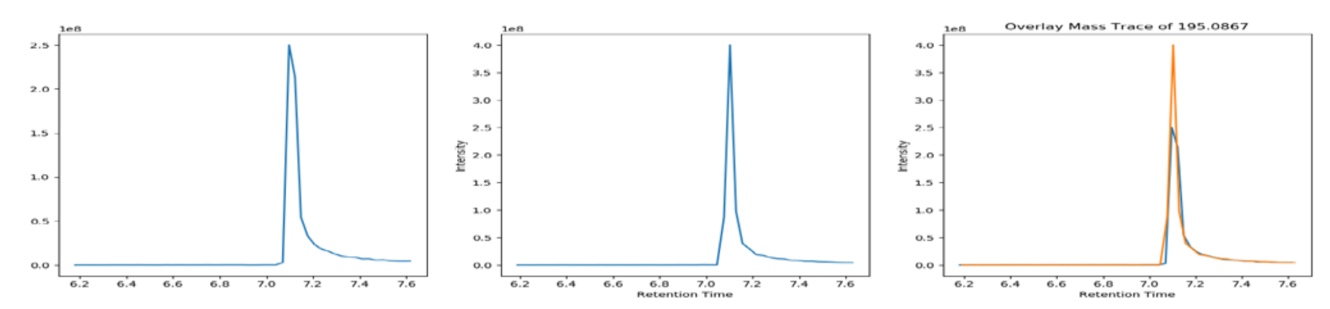
The driver program will always point to a directory of parquet-converted files. In the V 1.0 of Parsec we read all the scan data from all the files and then groupbykey(). Note that the groupbykey() operation is an expensive operation as it triggers a shuffling and transfer of data across the network as data is partitioned across nodes. However, in the context of the full Component Detection workflow this initial step is almost negligable

```
In [102]: #Filter blacktea_1.raw by BasePeakIntensity
          filteredScans = scans.rdd.filter(lambda s: s.FileName == 'blacktea_1.raw' and s.BasePeakIntensity > 1e7).take(5)
          for f in filteredScans:
              print(f.FileName, f.BasePeakIntensity)

          blacktea_1.raw 250080816.0
          blacktea_1.raw 213495824.0
          blacktea_1.raw 54395568.0
          blacktea_1.raw 33300088.0
          blacktea_1.raw 23759064.0

In [103]: #group the scans by keys which represents two raw files
          files = scans.rdd.map(lambda scan: pc.CreateKVpair(scan, 1)).groupByKey().take(5)
          print(files)

          [('blacktea_1.raw', <pyspark.resultiterable.ResultIterable object at 0x7ff9908d9dd8>), ('blacktea_2.raw', <pyspark.resultiterab
          le.ResultIterable object at 0x7ff9908d9a58>)]
```

CreateMassTracesSpark()

The CreateMassTracesSpark function takes an individual bin as input. First thing it does is create 2 sorted lists of the mz points. The first sorted list is by intensity. The second sorted list is by mass. It loops over the intensity sorted list, starting with the most intense. It then goes to the mass sorted list, and finds all other mz points within 10ppm (searching both higher and lower values). A scaling factor is applied to the 10ppm tolerance to make the tolerance larger on smaller mz values. All mz points that are within this tolerance are gathered into 1 mass trace object. These mz points are marked processed to not be used again. The output from the CreateMassTracesSpark is a key value pair. The key is the raw file name, the value is a list of mass traces that came from this bin.

```
In [ ]: #Get mass traces for all files. Note that the delegate function CreateKVpair() takes two arguments (scan and msOrder)
        #for demonstration we filter for caffeine (195.0867 M+H)
        traces = scans.rdd.map(lambda scan: pc.CreateKVpair(scan, 1)).groupByKey() \
        .flatMap(lambda file: pc.BinMassesSpark(file)) \
        .flatMap(lambda fileMassBin: pc.CreateMassTracesSpark(fileMassBin)) \
        .filter(lambda s: s[1].HighestIntensityMass > 195.07 and s[1].HighestIntensityMass < 195.1) \
        .collect()
```



Parallel Peak Detection

We are now ready to detect all the peaks from all traces from all files.

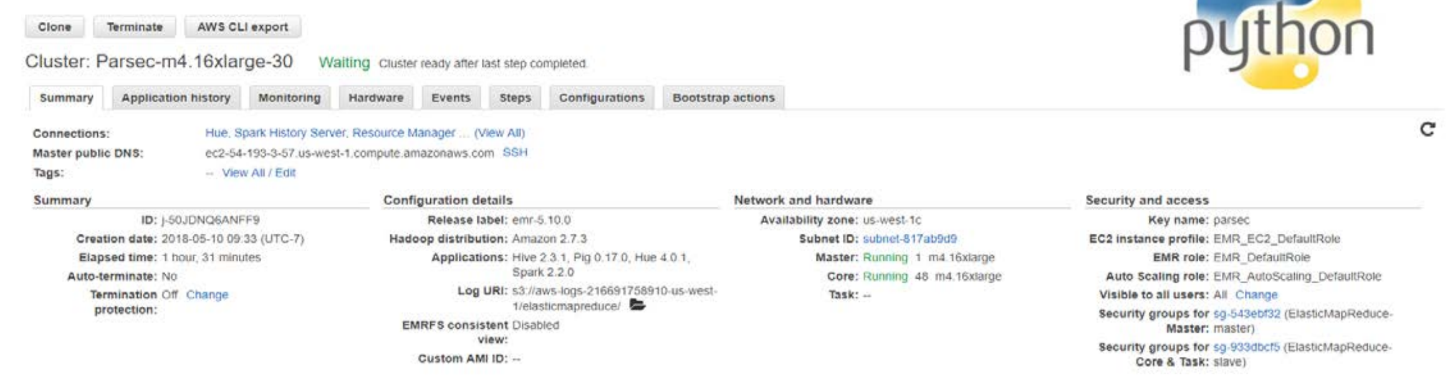
```
In [39]: peaks = scans.rdd.map(lambda scan: pc.CreateKVpair(scan, 1)).groupByKey() \
         .flatMap(lambda file: pc.BinMassesSpark(file)) \
         .flatMap(lambda fileMassBin: pc.CreateMassTracesSpark(fileMassBin)) \
         .flatMap(lambda trace: pc.MapCdl2PeakUtils(trace[1],5)).filter(lambda p: p.Intensity > 2e7).take(5)

In [40]: #print out all the detected peaks from all files
         print('Peaks:')
         print('FileID m/z Intensity RT')
         for p  in peaks:
             print(p.FileID,p.MZ,p.Intensity,p.RT)

         Peaks:
         FileID m/z Intensity RT
         blacktea_1.raw 138.06524658203125 101981904.0 7.0972566666666665
         blacktea_1.raw 195.0867156982422 250080816.0 7.0972566666666665
         blacktea_2.raw 138.0651092529297 161406496.0 7.103033333333332
         blacktea_2.raw 195.08657836914062 399965472.0 7.103033333333332
         blacktea_2.raw 196.0901641845703 28324078.0 7.103033333333332
```

An End-to-End Example for Feature Detection

```
In [24]: features = spark.read.parquet('data/*') \
         .rdd.map(lambda scan: pc.CreateKVpair(scan, 1)).groupByKey() \
         .flatMap(lambda file: pc.BinMassesSpark(file)) \
         .flatMap(lambda fileMassBin: pc.CreateMassTracesSpark(fileMassBin)) \
         .flatMap(lambda trace: pc.MapCdl2PeakUtils(trace[1], 1)) \
         .map(lambda p: pc.MapPeaks(p,0.1)).groupByKey() \
         .flatMap(lambda p: pc.DetectSmallMoleculeFeaturesFaster(p[1],50)) \
         .map(lambda f: pc.MapFeature(f)).persist().collect()
```

Performance Characterization

EMR Cluster



Clone | Terminate | AWS CLI export

Cluster: Parsec-m4.16xlarge-30 Waiting Cluster ready after last step completed.

Summary | Application history | Monitoring | Hardware | Events | Steps | Configurations | Bootstrap actions

Connections: Hue, Spark History Server, Resource Manager ... (View All)
Master public DNS: ec2-54-193-3-57.us-west-1.compute.amazonaws.com SSH
Tags: -- View All / Edit

Summary
ID: j-50JDNQ6ANFF9
Creation date: 2018-05-10 09:33 (UTC-7)
Elapsed time: 1 hour, 31 minutes
Auto-terminate: No
Termination protection: Off Change

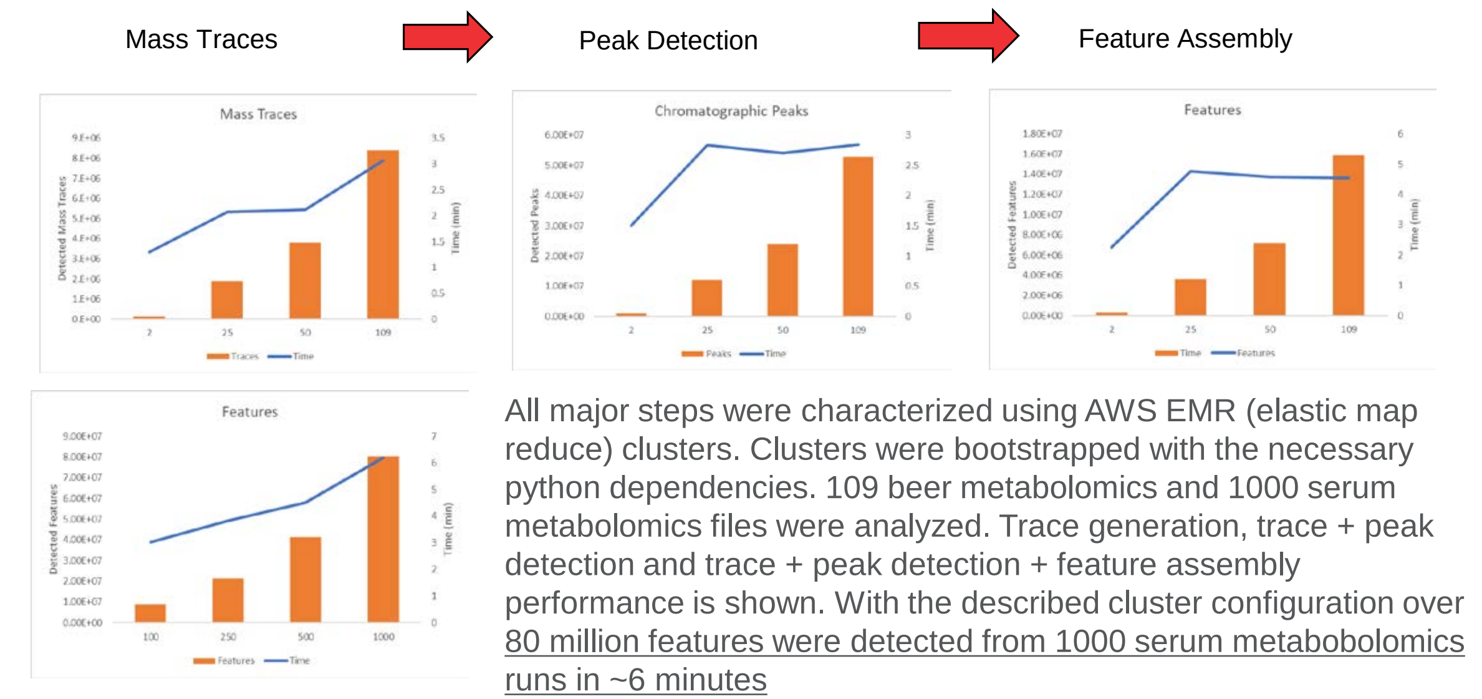
Configuration details
Release label: emr-5.10.0
Hadoop distribution: Amazon 2.7.3
Applications: Hive 2.3.1, Pig 0.17.0, Hue 4.0.1, Spark 2.2.0
Log URI: s3://aws-logs-216691758910-us-west-1/elasticmapreduce/
EMRFS consistent view: Disabled
Custom AMI ID: --

Network and hardware
Availability zone: us-west-1c
Subnet ID: subnet-817ab9d9
Master: Running 1 m4.16xlarge
Core: Running 48 m4.16xlarge
Task: --

Security and access
Key name: parsec
EC2 instance profile: EMR_EC2_DefaultRole
EMR role: EMR_DefaultRole
Auto Scaling role: EMR_AutoScaling_DefaultRole
Visible to all users: All Change
Security groups for Master: sg-543ebf32 (ElasticMapReduce-master)
Security groups for Core & Task: sg-933dbcf5 (ElasticMapReduce-slave)

Figure 5. Cluster Configuration.

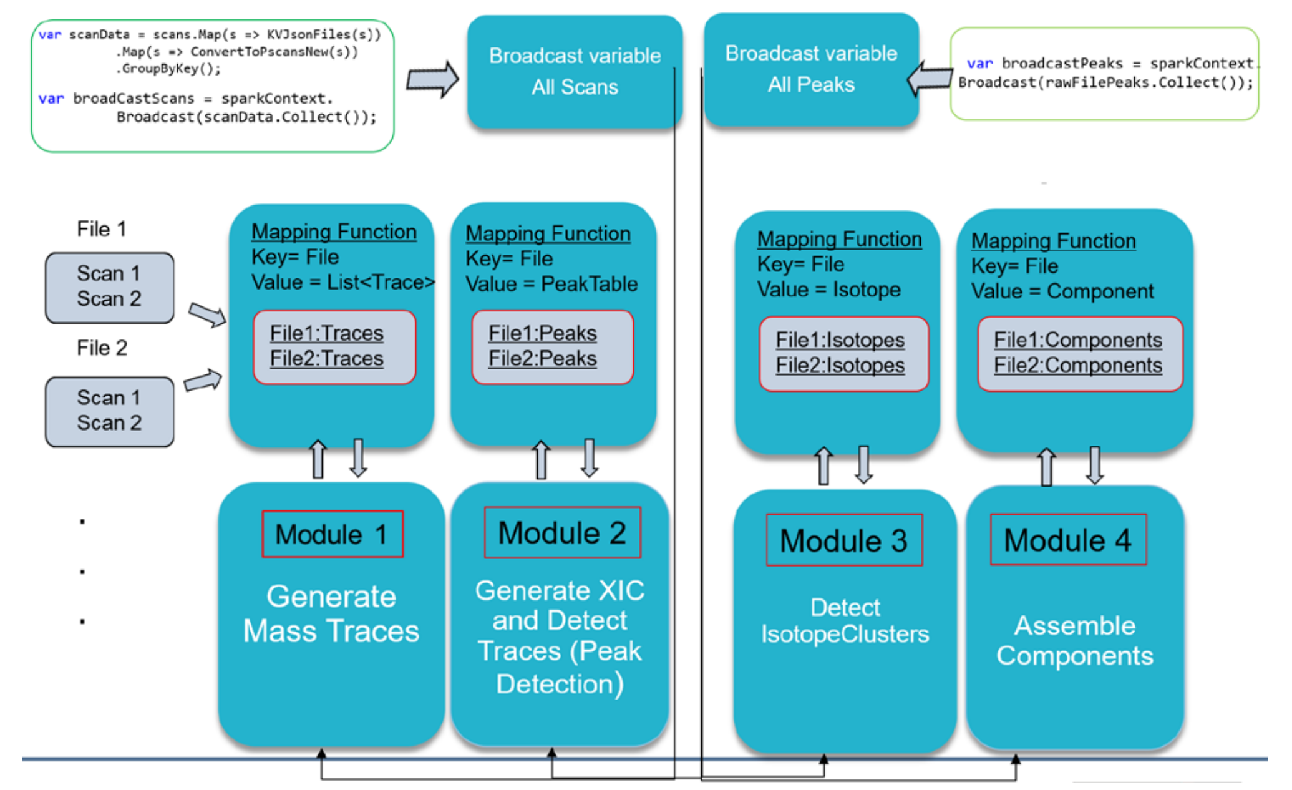
spark-submit --deploy-mode cluster --driver-memory 8g --executor-memory 8g --num-executors 1243 --executor-cores 5 run.py s3n://parsecdata/parquetfiles/* s3n://parsecdata/results/output



All major steps were characterized using AWS EMR (elastic map reduce) clusters. Clusters were bootstrapped with the necessary python dependencies. 109 beer metabolomics and 1000 serum metabolomics files were analyzed. Trace generation, trace + peak detection and trace + peak detection + feature assembly performance is shown. With the described cluster configuration over <u>80 million features were detected from 1000 serum metabobolomics runs in ~6 minutes</u>

Figure 6. Performance evaluation.

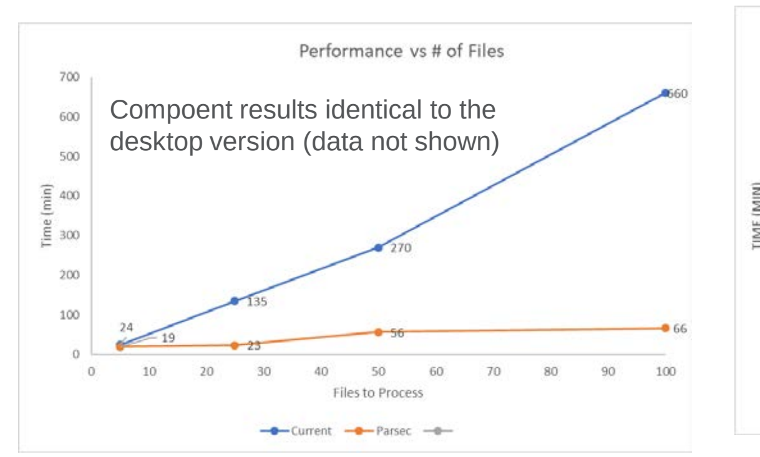
Legacy Algorithm Adaption

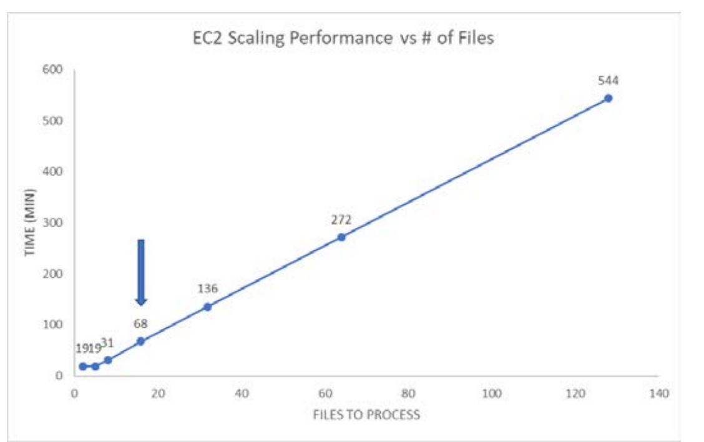


```
var scanData = scans.Map(s => KVJsonFiles(s))
        .Map(s => ConvertToPscansNew(s))
        .GroupByKey();

var broadCastScans = sparkContext.
        Broadcast(scanData.Collect());
```

```
var broadcastPeaks = sparkContext.
Broadcast(rawFilePeaks.Collect());
```

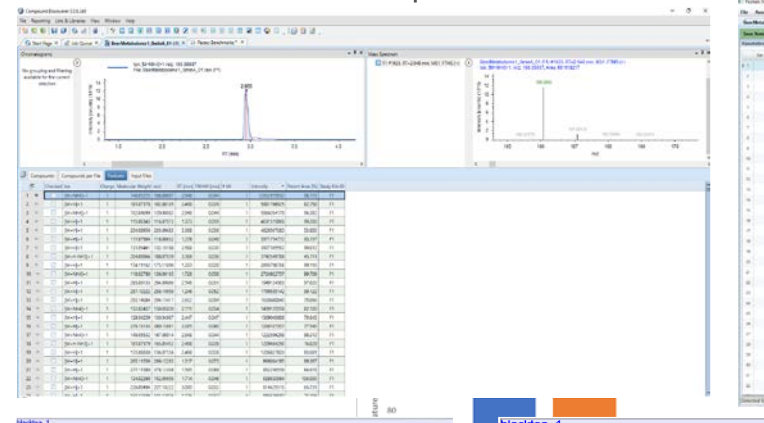
Broadcast variable All Scans
Broadcast variable All Peaks
File 1
Scan 1 Scan 2
File 2
Scan 1 Scan 2
Mapping Function Key= File Value = List<Trace>
File1:Traces File2:Traces
Mapping Function Key= File Value = PeakTable
File1:Peaks File2:Peaks
Mapping Function Key= File Value = Isotope
File1:Isotopes File2:Isotopes
Mapping Function Key= File Value = Component
File1:Components File2:Components
Module 1
Generate Mass Traces
Module 2
Generate XIC and Detect Traces (Peak Detection)
Module 3
Detect IsotopeClusters
Module 4
Assemble Components



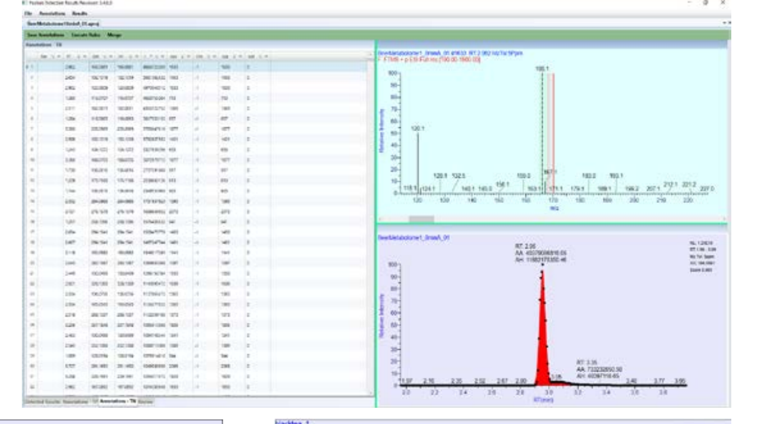


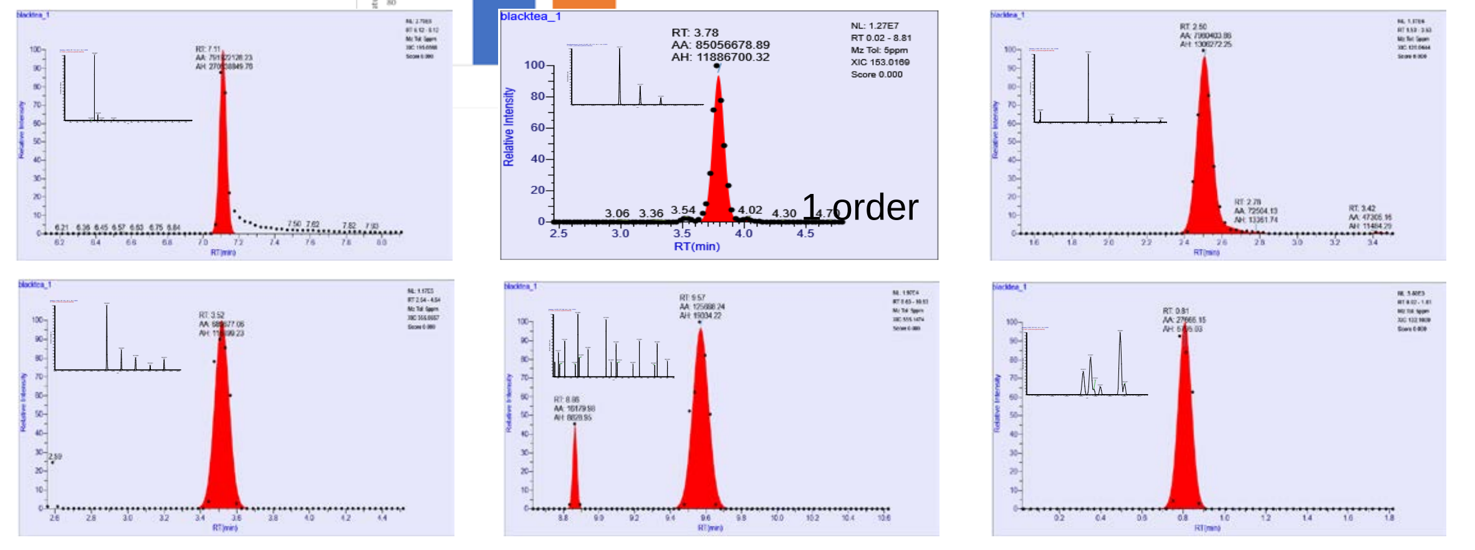
Preliminary Qualitative Analysis

Thermo Scientific™ Compound Discoverer™



Parsec Annotation Viewer/Editor





CONCLUSIONS

- LCMS raw data was transformed to a distributed parquet format for parallel ingestion and high performance processing. A map-reduce programming model was successfully applied to a legacy desktop algorithm. In addition a new prototype algorithm was also developed fully paralyzing computational tasks at the scan level.
- We demonstrated the same code can be scaled across a cluster compute node using the AWS EMR platform with near constant time. We show POC by detecting 80 million features across 1000 metabolomics LCMS runs in ~6 minutes.
- Qualitative analysis shows reasonable overlap between our existing desktop component detection algorithm at the feature level. All legitimate features > 1e6 intensity in test files detected in Thermo Scientific™ Compound Discoverer™ software were also detected in the prototype version of Parsec. A qualitative review across the dynamic range indicates the presence of detectable features down 5 orders of magnitude. The functional programming model enables interchangeable peak detection/isotope clustering techniques to be incorporated with little modification to the execution code or framework.

Future Work

Our initial goal was to demonstrate a new scalable programming model applied to the LCMS domain. Future work will include modification/replacement of the delegate detection functions while verifying sensitivity/specificity using precision recall curves on well-annotated big data sets. Related activities will also include streaming scan data and improving the file conversion/upload bottlenecks ensuring a performant end-to end user experience enabling large scale statistical power

REFERENCES

1. Dean et al., MapReduce: Simplified Data Processing on Large Clusters. ODSI 2004.
2. Tautenhahn et al., Highly sensitive feature detection for high resolution LC/MS Bioinformatics, 2008

ACKNOWLEDGEMENTS

We would like to thank the Thermo Fisher CRC committee for funding this research effort as well as Swapnil Ahuja for project support activities.

TRADEMARKS/LICENSING