Practical Course in Bioinformatics

Algorithm Group – Final Presentation

Max Berrendorf Thomas Lange Tina Raissi Matthias Voit

ZKF Research Group Computational Biology and Bioinformatics
Helmholtz Institute for Biomedical Engineering
RWTH University Hospital

Outline

Section 1

The BigWig Format

The BigWig format

..enhances the Wig format:

- Both are used to store genomic signals
- A BigWig file is a compressed binary indexed file
- Most importantly, it allows remote random access

..uses multiple software layers (written in c):

1. Data transfer layer
2. URL data cache layer
3. Indexing
4. Compression

Data transfer layer

Existing web–based protocols are used:

- HTTP/HTTPS accept byte–ranges since HTTPS/1.1
- Protocols associated with resuming interrupted FTP transfers
- OpenSSL provides SSL support for HTTPS via the BIO protocol

URL data cache layer

Since files are typically viewed many times without changing:

- Data is fetched in blocks of 8 Kb and each block is cached
- Implemented by using two files for each file cached; a bitmap file and a data file
- The data file is a Linux sparse file

Efficiency gains mainly as:

- The file header and the root block of the index are always accessed
- One might look at the same region for several times

Indexing

The indexing is based on R trees:

- Data is stored only in the leaf nodes
- The index nodes contain the span of all child nodes

They perform well for overlapping search intervals (here: genomic regions)

Only every 512th item is indexed: index less than 1% of the data

Compression

- Regions between indexed items (512 items) are seperately compressed
- Deflation techniques are similar to those used by gzip

The final layer is responsible for fetching and decoding blocks specified by the index

Section 2

Random Access to Remote BigWig Files

Two different libraries to stream BigWig files

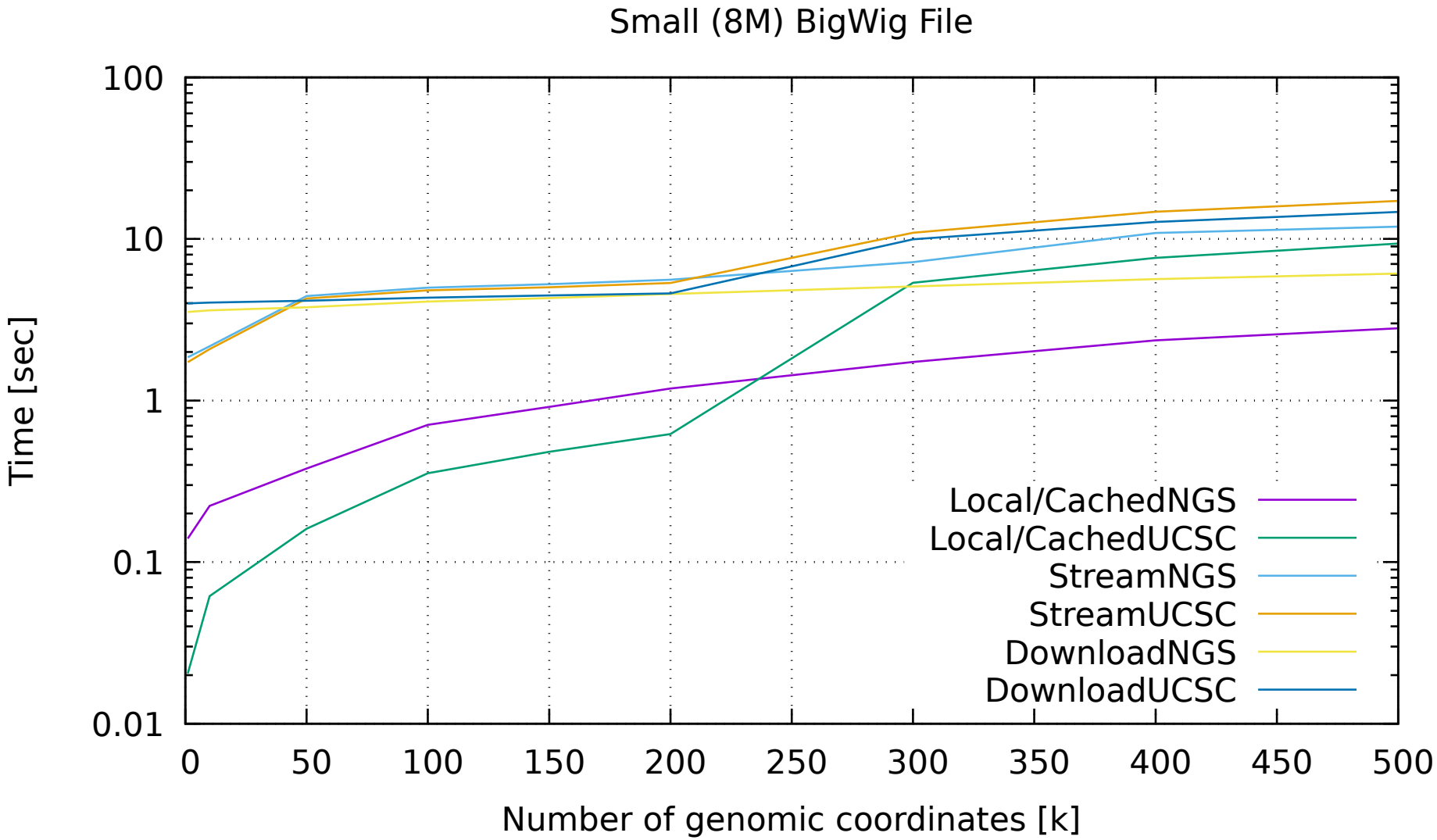
"UCSC":

- Does not allow arbitrary access to a BigWig file
- Only five predefined functions are available

NGSLib:

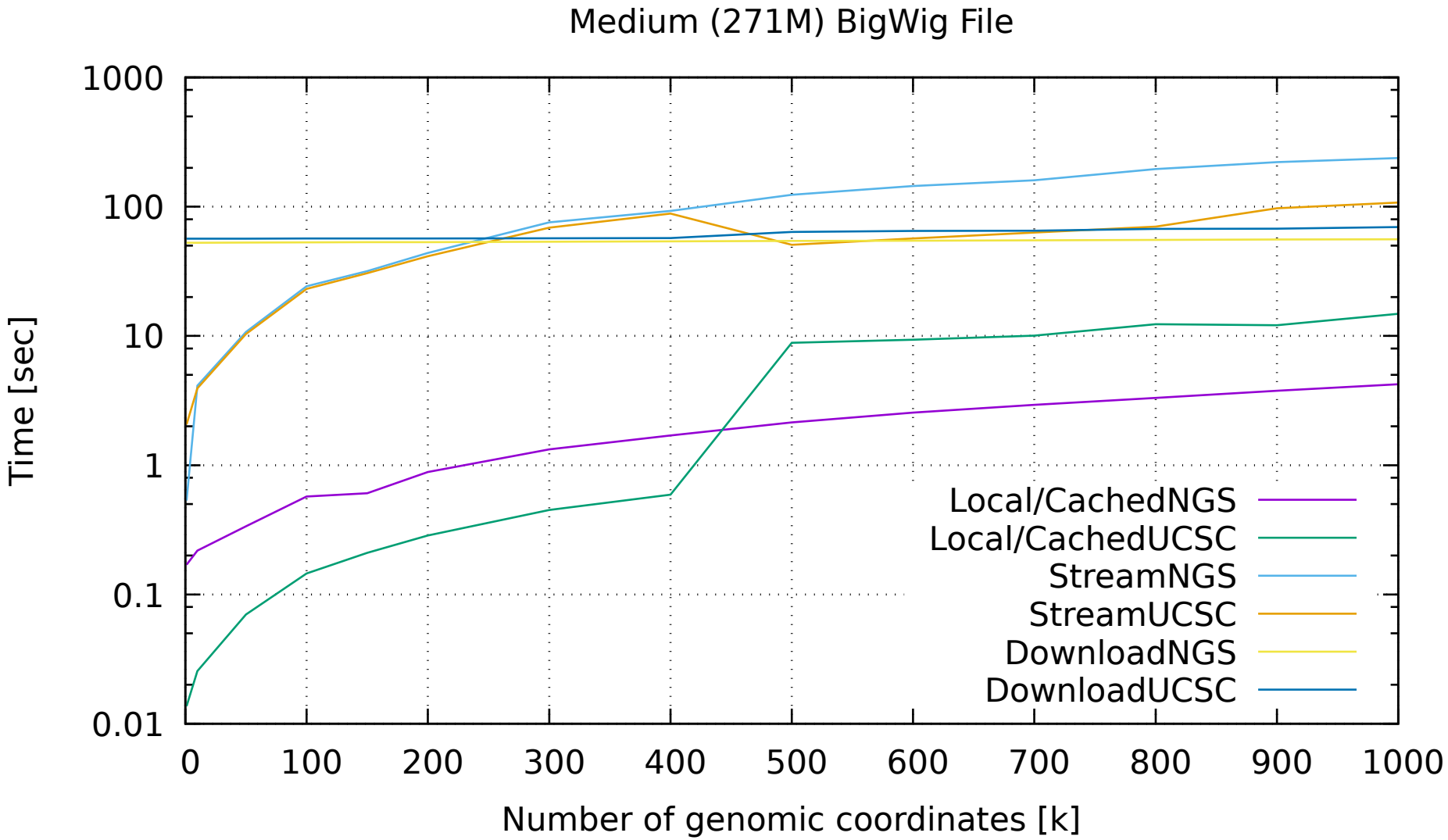
- Provides full flexibility
- Any specified regions of a (remote) BigWig file can be fetched

Averages over a small BigWig file



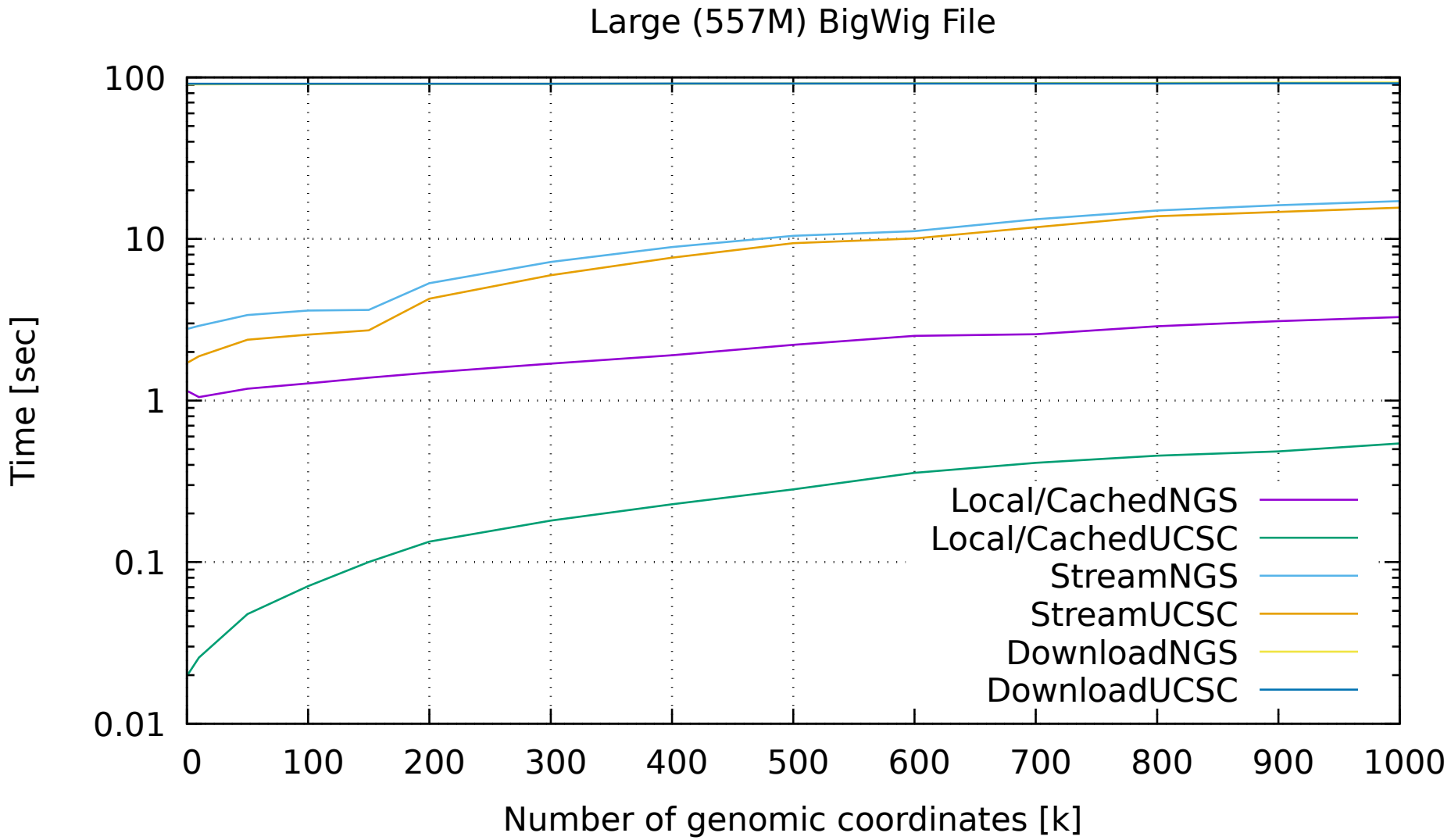
(Filename: wgEncodeHaibTfbsA549Atf3V0422111Etoh02RawRep1.bigWig)

Averages over a medium BigWig file



(Filename: wgEncodeHaibTfbsK562Cebpdsc636V0422111RawRep2.bigWig)

Averages over a large BigWig file



(Filename: wgEncodeHaibTfbsGm12878Pol24h8Pcr1xRawRep2.bigWig)

Section 3

Improving Intersection Test and Motif Analysis

Intersect test

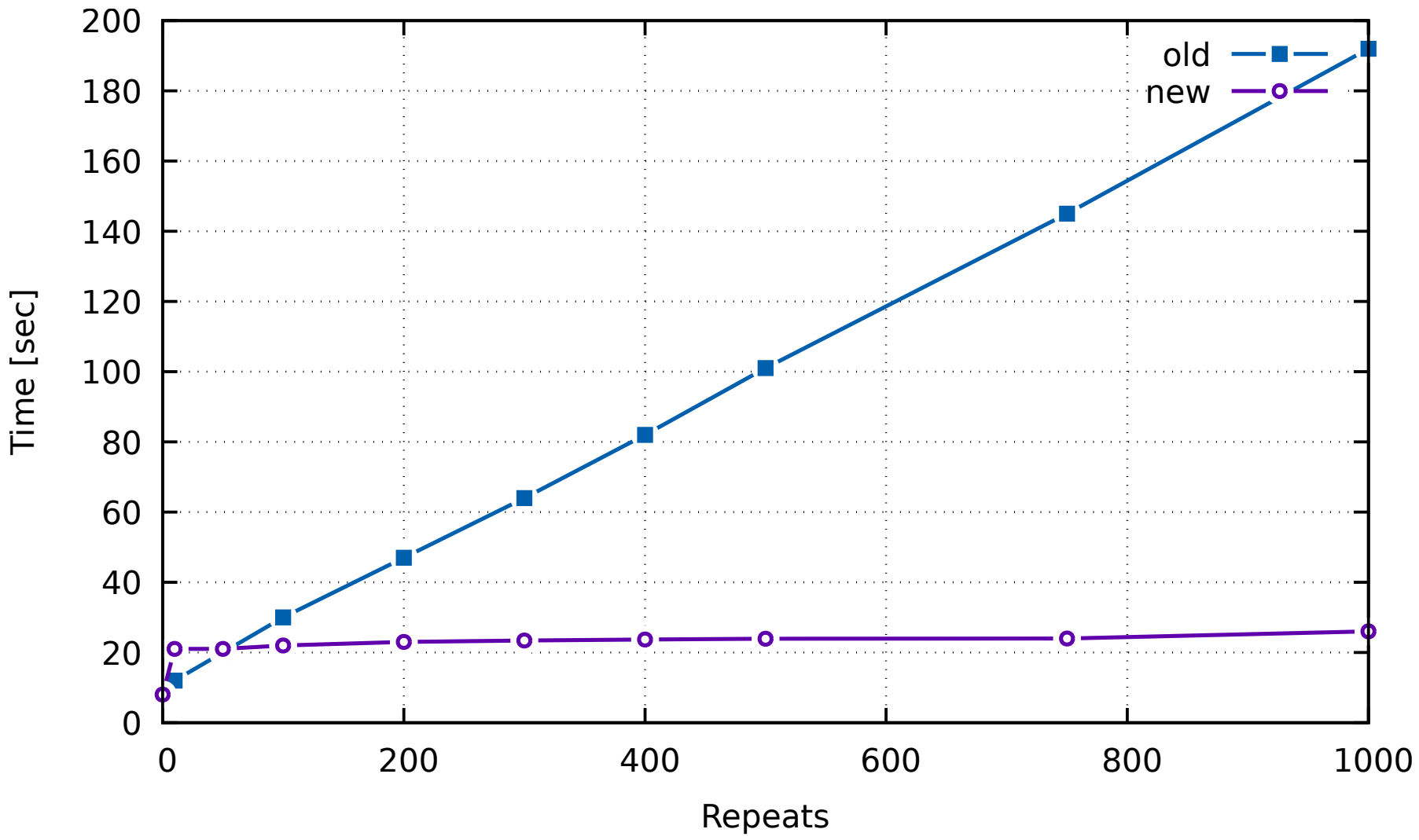
- Randomization by shuffling
- Combine GenomicRegionSets a, b

$$[region\_a_1, \cdots, region\_a_n, region\_b_1, \cdots, region\_b_m]$$

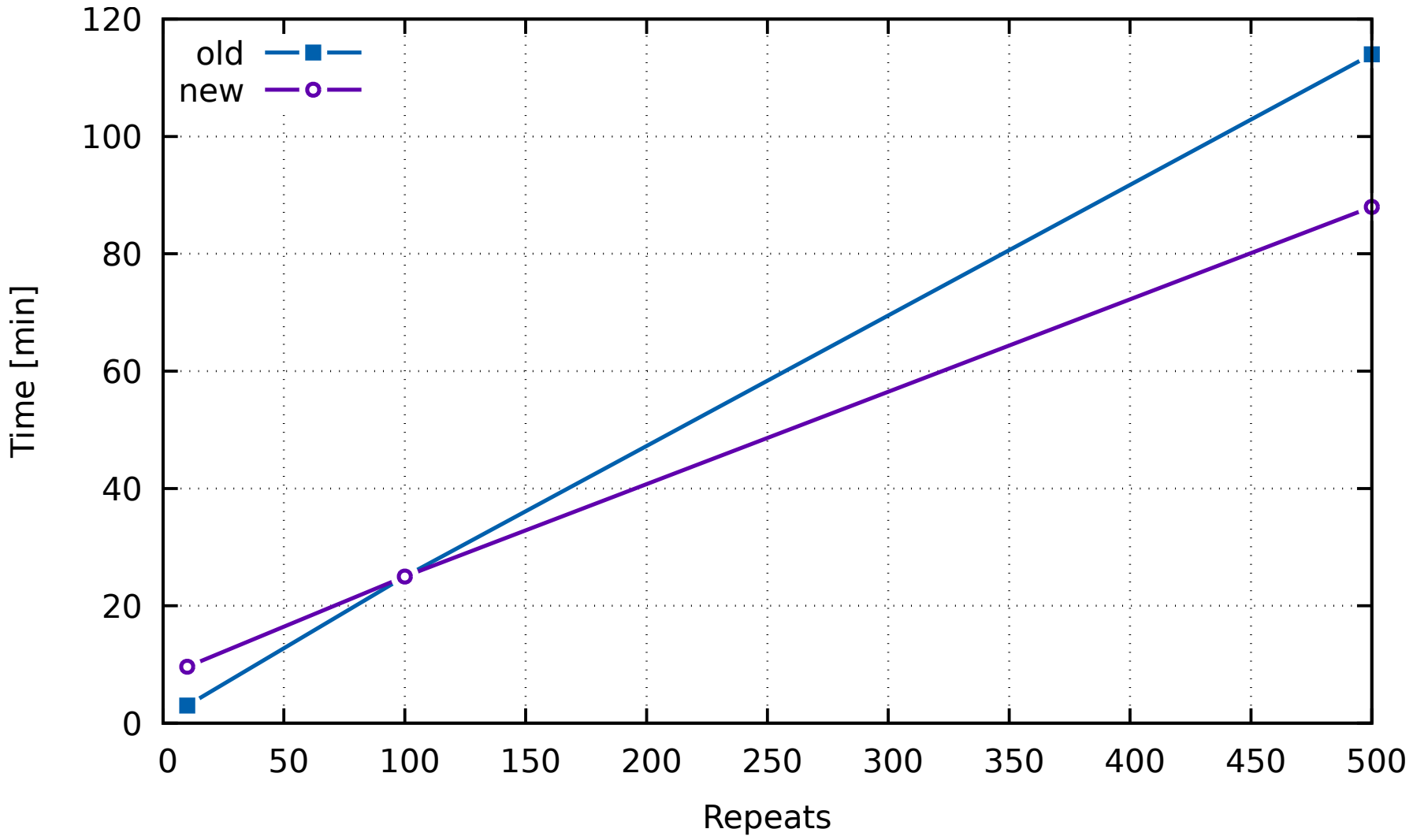
- If amount of repeats is too large, use memoization

	$region\_b_1$		$\cdots$		$region\_b_n$
$region\_a_1$	1	0	0	0	0
	0	0	0	0	0
$\vdots$	0	0	0	0	1
	0	1	0	0	0
$region\_a_n$	0	0	1	1	0

query: PU1 len=213744
reference: chr19 of cDC_H3K4me3_peaks len=706



query: PU1 len=213744
reference: cDC_H3K4me3_peaks len=20796



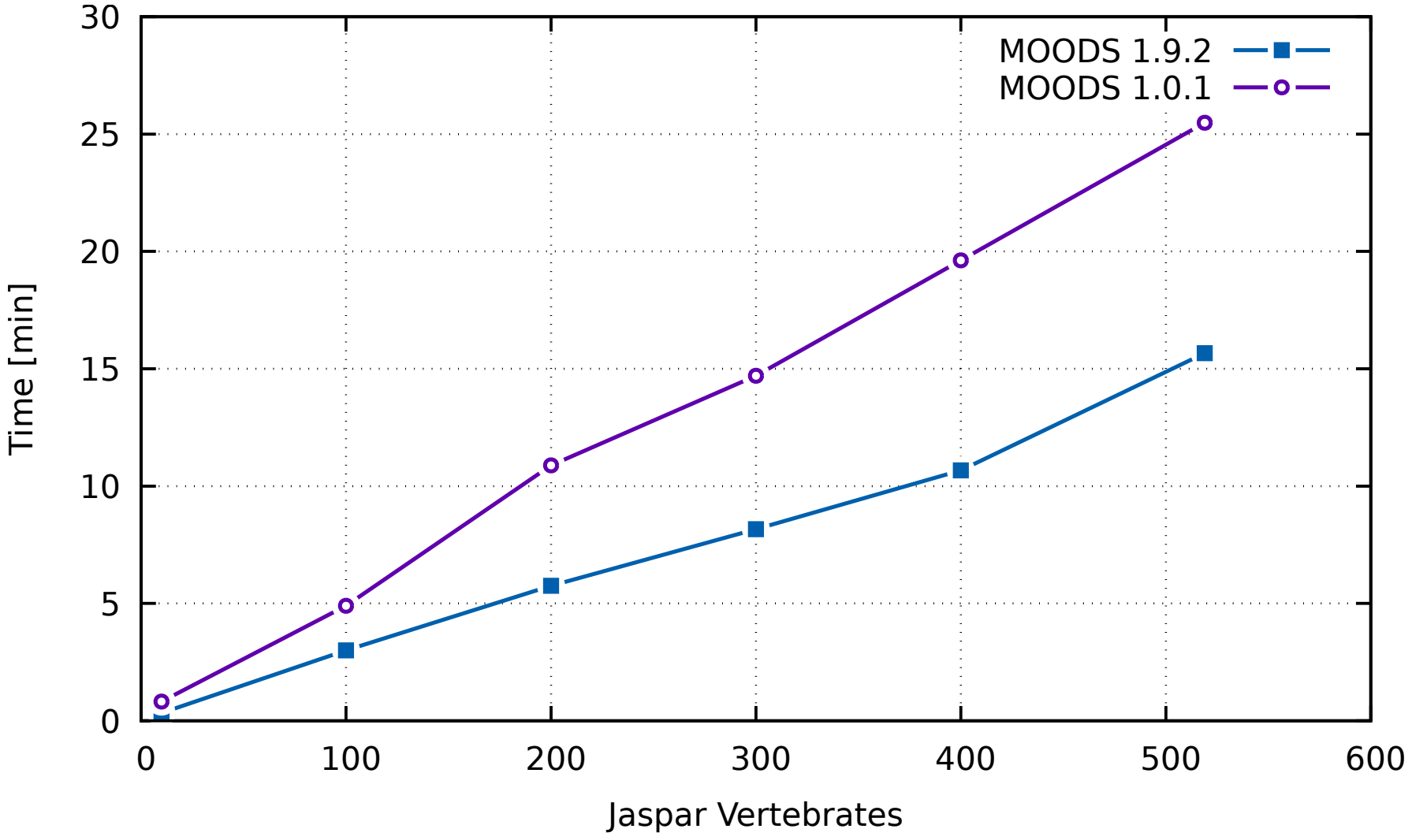
Motif Matching

- match position weight matrices (PWM) against DNA sequences
- find putative binding sites in the input regions and random regions
- MOODS 1.0.1 search function
- MOODS 1.9.2 scanner object with scan function

Motif Occurence Detective Suite

- lookahead filtration algorithm (LF)
- find the statistically most significant submatrix of the PWM (scanning window)
- scan DNA with a finite state automaton that finds subsequences that score well against the scanning window
- full score only evaluated at these sequence positions

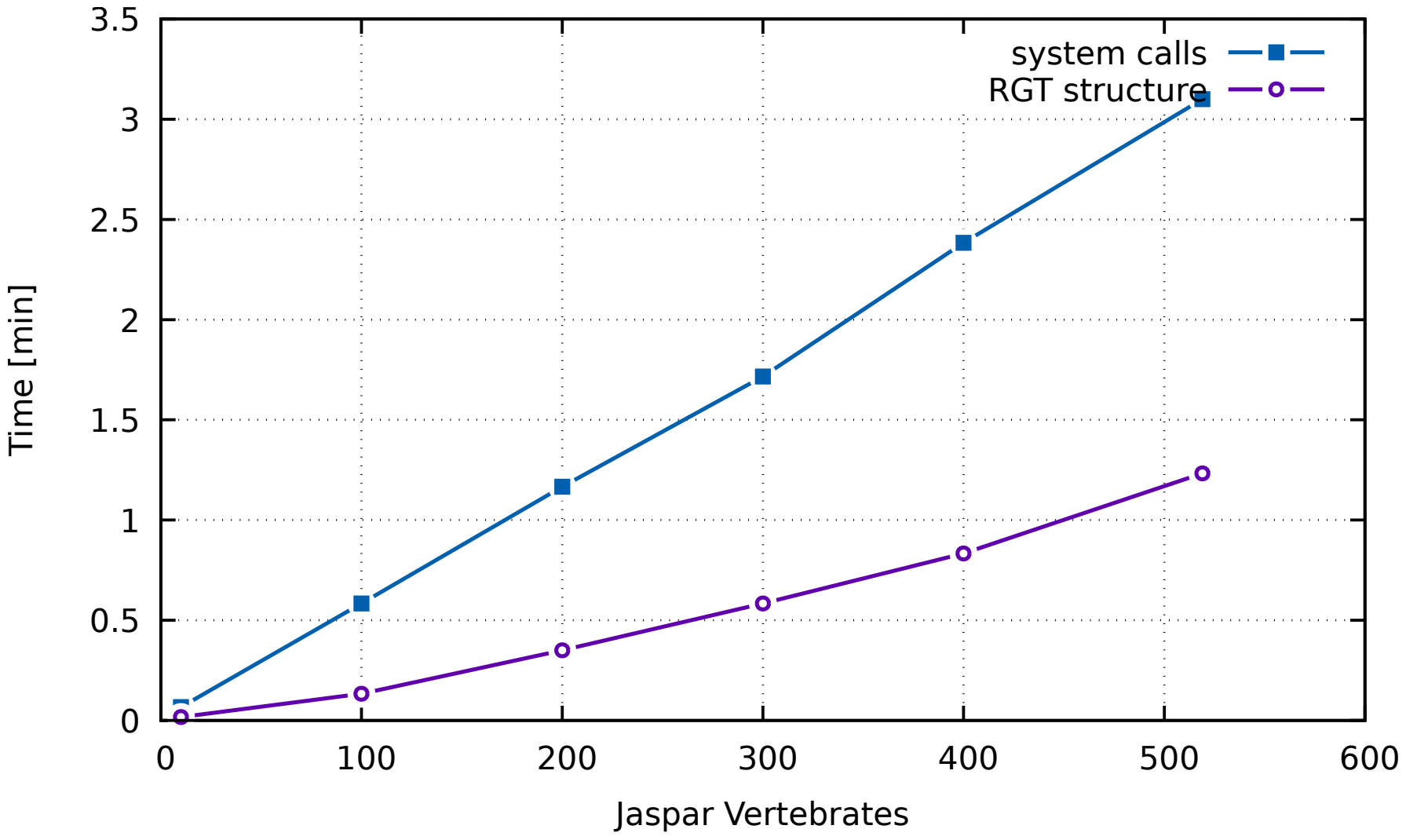
Motif Matching
mm9 – CDP_PU1_peak, rand-proportion=2



Motif Enrichment

- verify which transcription factors are enriched in input regions with fishers exact test
- count the number of putative binding sites inside the input and random regions
- perform a statistical test to determine which transcription factors are more likely to be enriched

Motif Enrichment
mm9 – CDP_PU1_peak, matching with MOODS 1.0.1



Section 4

Outsourcing Computation to C

Problem

Problem

- Framework is written in Python
- Computation in C is orders of magnitude faster
- How can compute-intensive tasks be delegated to C?

Solution

- Use `ctypes` library [a] offering
  - C compatible data types:
    `int` $\rightsquigarrow$ `c_int`, `double` $\rightsquigarrow$ `c_double`, ...
  - Calling C-functions in shared libraries from pure Python

[a] `https://docs.python.org/2/library/ctypes.html`

Porting functionality to C

Rewritten the following functions:

With interface to Python

- `GenomicRegionSet.py:intersect`
- `GenomicRegionSet.py:jaccard`

Pure C

- `GenomicRegion.py:overlap`
- `GenomicRegion.py:__cmp__`

Binding shared library

Added to initialisation of GenomicRegionSet.py

```
from ctypes import *

# Determine path of shared library
me = os.path.abspath(os.path.dirname(__file__))
lib = cdll.LoadLibrary(os.path.join(me, "..", "librgt.so"))

# Bind library
ctypes_jaccardC = lib.jaccard

# Specify data types
ctypes_jaccardC.argtypes = [POINTER(c_char_p), POINTER(c_int),
    POINTER(c_int), c_int, POINTER(c_char_p), POINTER(c_int),
    POINTER(c_int), c_int]
ctypes_jaccardC.restype = c_double
```

Replacing Python methods by wrappers

Wrapper for `jaccard` and `intersect`

- Introduce flag `use_c`:
  - `use_c = True` $\rightsquigarrow$ use C for computation.
  - `use_c = False` $\rightsquigarrow$ use existing Python implementation.

Method `jaccard`

```
def jaccard(self, query, use_c=True):
        if use_c:
            return self.jaccard_c(query)
        else:
            return self.jaccard_python(query)
```

Calling the shared library

Method

- Convert native Python datatypes to C-compatible types
- Delegate functionality to C-library

Method jaccard_c

```
# Convert to ctypes
chroms_self_python = [gr.chrom for gr in self.sequences]
chroms_self_c = (c_char_p * len(chroms_self_python))(*
    chroms_self_python)

[...]

# Call C-function
return ctypes_jaccardC(chroms_self_c, initials_self_c,
    finals_self_c, len(self), chroms_query_c, initials_query_c,
    finals_query_c, len(query))
```

Digression: Algorithmic optimisations

Algorithmic optimisation of `jaccard_c`

- Only size of intersection interesting $\rightsquigarrow$ avoid constructing the result `GenomicRegionSet`.
- Size of union can be computed without computing the union: $|A \cup B| = |A| + |B| - |A \cap B|$.

Pseudocode

1. Compute sizes of both sets.
2. Compute size of intersection.
3. Compute size of union via $|A \cup B| = |A| + |B| - |A \cap B|$.
4. Return jaccard index: $|A \cap B| / |A \cup B|$.

Section 5

Minimising Deepcopies

Problem

Problem

- `GenomicRegionSet` passed as parameter: reference vs new object
- deepcopy is time consuming

Solution

- Flag to indicate whether to use by reference or create new object
- In the latter case create new objects and append regions

Changes

Overview of changes

- New flag `w_return` in `extend` of `GenomicRegionSet` and `GenomicRegion`.
- Use pre-existing flags `w_return` and `output` of `merge` and `combine` in `GenomicRegionSet`.
- New boolean attribute `merged` in class `GenomicRegionSet`
- Modifications in implementation of 8 functions
- Only one call to deepcopy in `filter_by_gene_association` in the final version of `GenomicRegionSet.py`

Changes: Modified functions

GenomicRegion

- extend

GenomicRegionSet

- extend
- combine
- intersect_python
- intersect_count
- merge
- cluster
- window

Extend function

Old version: GenomicRegionSet

```
def extend(self, left, right, percentage=False):
    if percentage: #some code
    else:
        for s in self.sequences:
            s.extend(left, right)
```

Old version: GenomicRegion

```
def extend(self, left, right):
    self.initial -= left
    self.final += right
    #if left, right are negative, switching the border may be
        necessary
    if self.initial > self.final:
        self.initial, self.final = self.final, self.initial
    self.initial = max(self.initial, 0)
```

New Version: GenomicRegion

```
def extend(self, left, right, w_return=False):
    if w_return:
        z = GenomicRegion(chrom=self.chrom, initial=self.initial -
            left, final=self.final + right, name=self, orientation=
            self.orientation, data=self.data)
    else:
        z = self
        z.initial -= left
        z.final += right


    # if left, right are negative, switching the border may be
        necessary
    if z.initial > z.final:
        z.initial, z.final = z.final, z.initial
    z.initial = max(z.initial, 0)

    return z
```

New Version: GenomicRegionSet

```
def extend(self, left, right, percentage=False, w_return=False):
  z = GenomicRegionSet(name=self.name)
  if percentage:
   if percentage > -50:
    for s in self.sequences:
     if w_return:
      z.add(s.extend(int(len(s) * left / 100), int(len(s) * right /
          100), w_return=True))
      else:
       s.extend(int(len(s) * left / 100), int(len(s) * right /
           100))
  else:
   for s in self.sequences:
    if w_return:
     z.add(s.extend(left, right, w_return=True))
    else:
     s.extend(left, right)
  if w_return: return z
  else: return
```

Side effects of extend's change: window

old

```
def window(self,y,adding_length = 1000):
        #some code
        extended_self = deepcopy(self)
        extended_self.extend(adding_length,adding_length)
        #other code
```

new

```
def window(self,y,adding_length = 1000):
        #some code
        extended_self = self.extend(adding_length,adding_length,
            w_return = True)
        #other code
```

Side effects of extend's change: cluster

old

```
def cluster(self,max_distance):
    #some code
    else:
        #some code
        for s in self.sequences[1:]:
            s_ext = deepcopy(s) #deepcopy in a loop!
            s_ext.extend(max_distance,max_distance)
```

new

```
def cluster(self,max_distance):
    #some code
    else:
        #some code
        for s in self.sequences[1:]:
            s_ext = s.extend(max_distance, max_distance, w_return=
                True)
```

Setting the boolean attribute merged

old

```
def merge(self, w_return=False, namedistinct=False, strand_specific
    =False):
    #some code
    z = GenomicRegionSet(name=self.name)
    #some code
    if w_return: return z
    else: self.sequences = z.sequences
```

new

```
def merge(self, w_return=False, namedistinct=False, strand_specific
    =False):
    #some code
    z = GenomicRegionSet(name=self.name)
    #some code
    z.merged = True
    if w_return:
        return z
    else: self.sequences = z.sequences
```

Using flag merged: intersect

old

```
def intersect(self, y, ...):
  #some code
  a = copy.deepcopy(self)
  b = copy.deepcopy(y)
  if not a.sorted: a.sort()
  if not b.sorted: b.sort()
  if mode == OverlapType.
      OVERLAP:
    a.merge()
    b.merge()
```

new

```
def intersect(self, y, ...):
  #some code
  a = self
  b = y
  if not a.sorted: a.sort()
  if not b.sorted: b.sort()
  if mode == OverlapType.
      OVERLAP:
    if not a.merged: a = a.
        merge(w_return=True)
    if not b.merged: b = b.
        merge(w_return=True)
```

Using flag merged: intersect_count

old

```
def intersect_count(self, regionset, ...):
    a = copy.deepcopy(self)
    b = copy.deepcopy(regionset)
    #some code
        if not a.sorted: a.sort()
        if not b.sorted: b.sort()
        a.merge()
        b.merge()
```

new

```
def intersect_count(self, regionset, ...):
        a = self
        b = regionset
        if not a.sorted: a.sort()
        if not b.sorted: b.sort()
        if not a.merged: a = a.merge(w_return=True)
        if not b.merged: b = b.merge(w_return=True)
```

Two versions of combine

old

```
def combine(self, region_set, change_name=True, output=False):
    if output:
        a = copy.deepcopy(self)
        a.sequences.extend(region_set.sequences)
        if change_name: #other codes
    else: #other code
        self.sequences.extend(region_set.sequences) #other code
```

new

```
def combine(self, region_set, change_name=True, output=False):
    if output:
        a = GenomicRegionSet(name="")
        for s in self.sequences: a.add(s)
        for s in region_set.sequences: a.add(s)
        if change_name: #other codes
        return a
    else: #other code
```

Side effects of combine and intersect's changes: jaccard

old

```
def jaccard(self,query):
    a = copy.deepcopy(self)
    b = copy.deepcopy(query)
    #some code
    intersects = a.intersect(b)
    intersects.merge()
    inter = intersects.total_coverage()
    a.combine(b, change_name=False)
    a.merge() #some other code
```

Side effects of combine and intersect's changes: jaccard

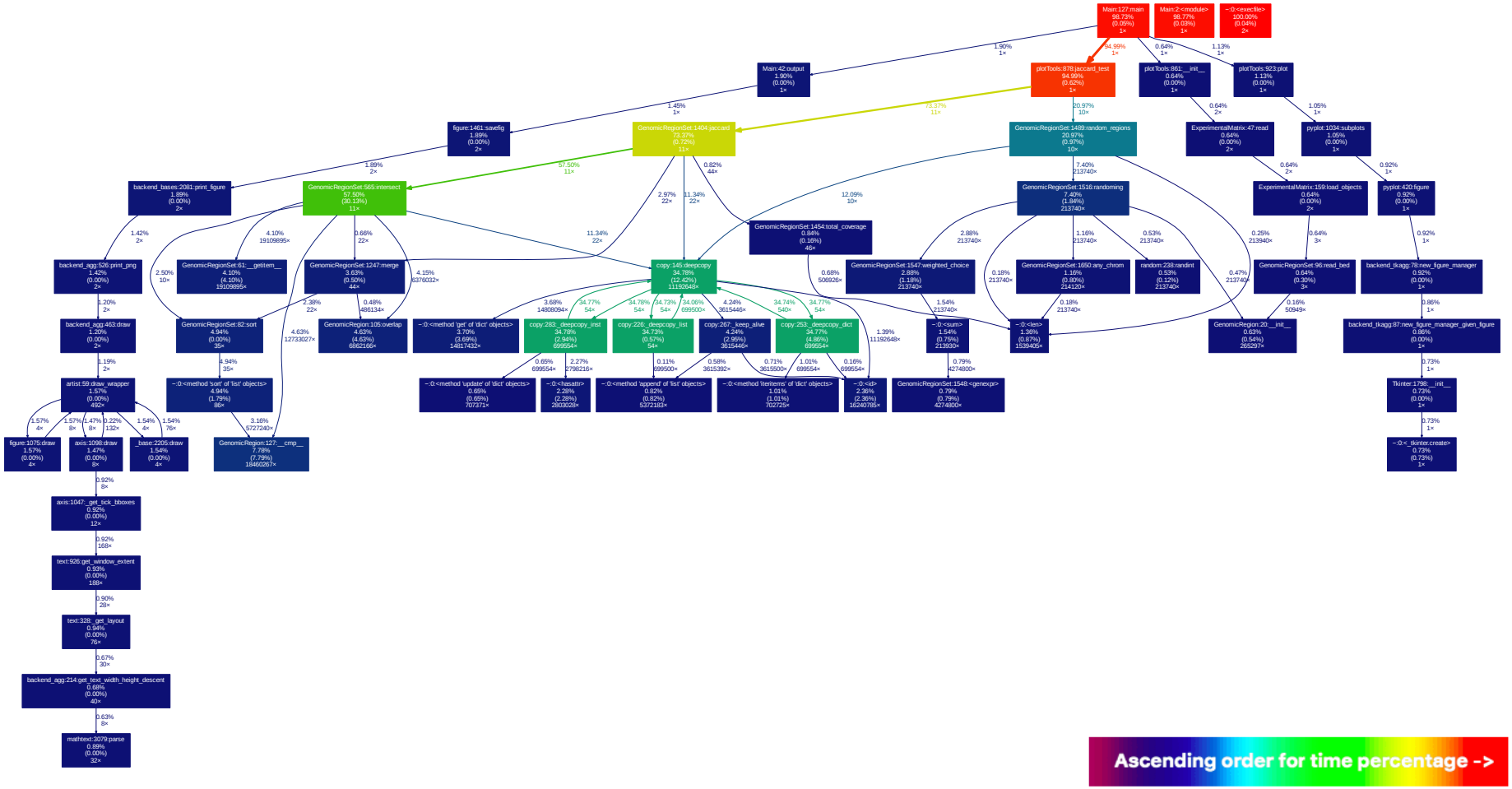
new

```
def jaccard_python(self, query):
    b = query
    #some code
    intersects = self.intersect(b)
    intersects.merge()
    inter = intersects.total_coverage()
    a = self.combine(b, change_name=False, output=True)
    if not a.merged: a.merge()
```

Section 6

Results

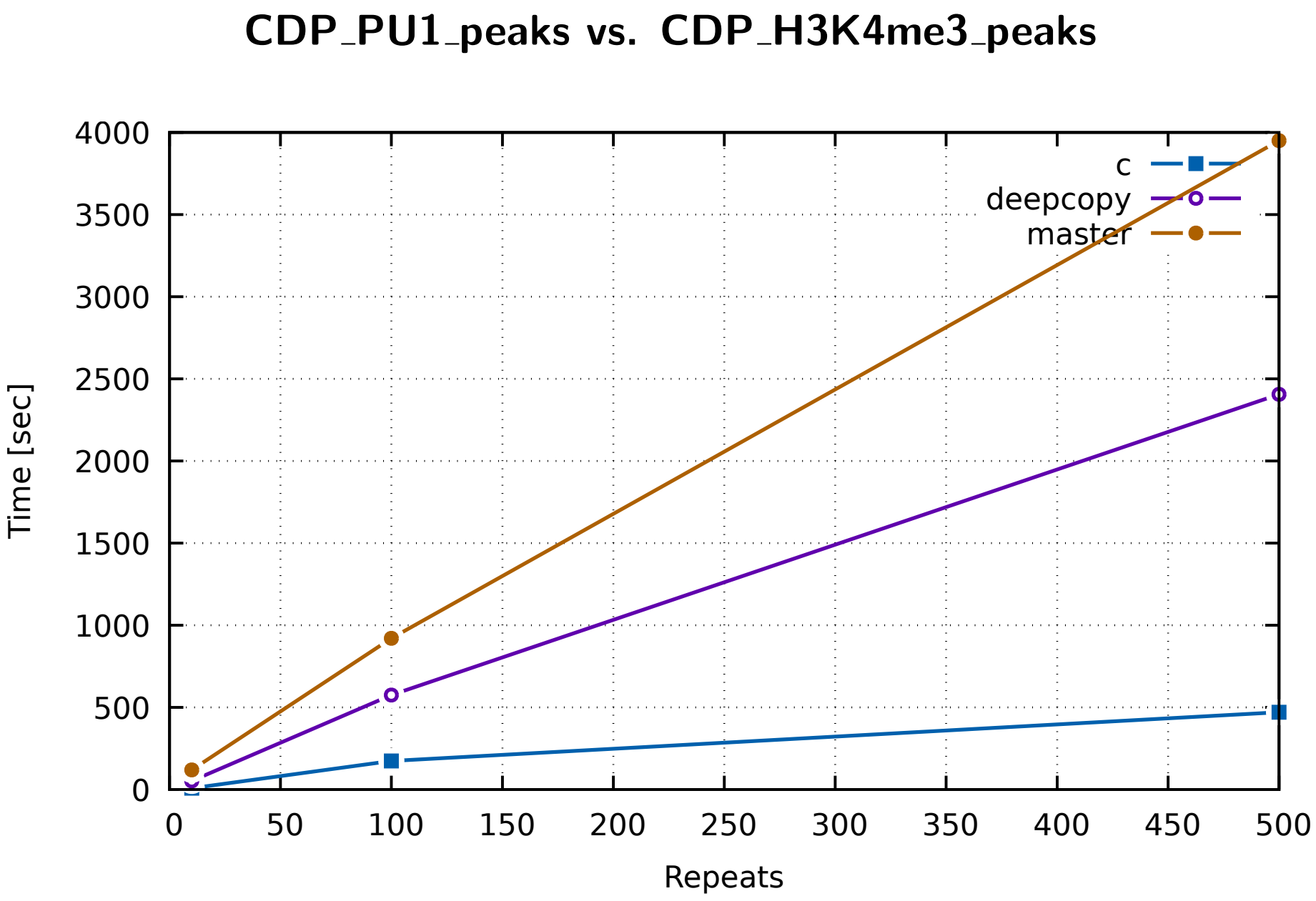
Profiling Jaccard Test with deepcopies



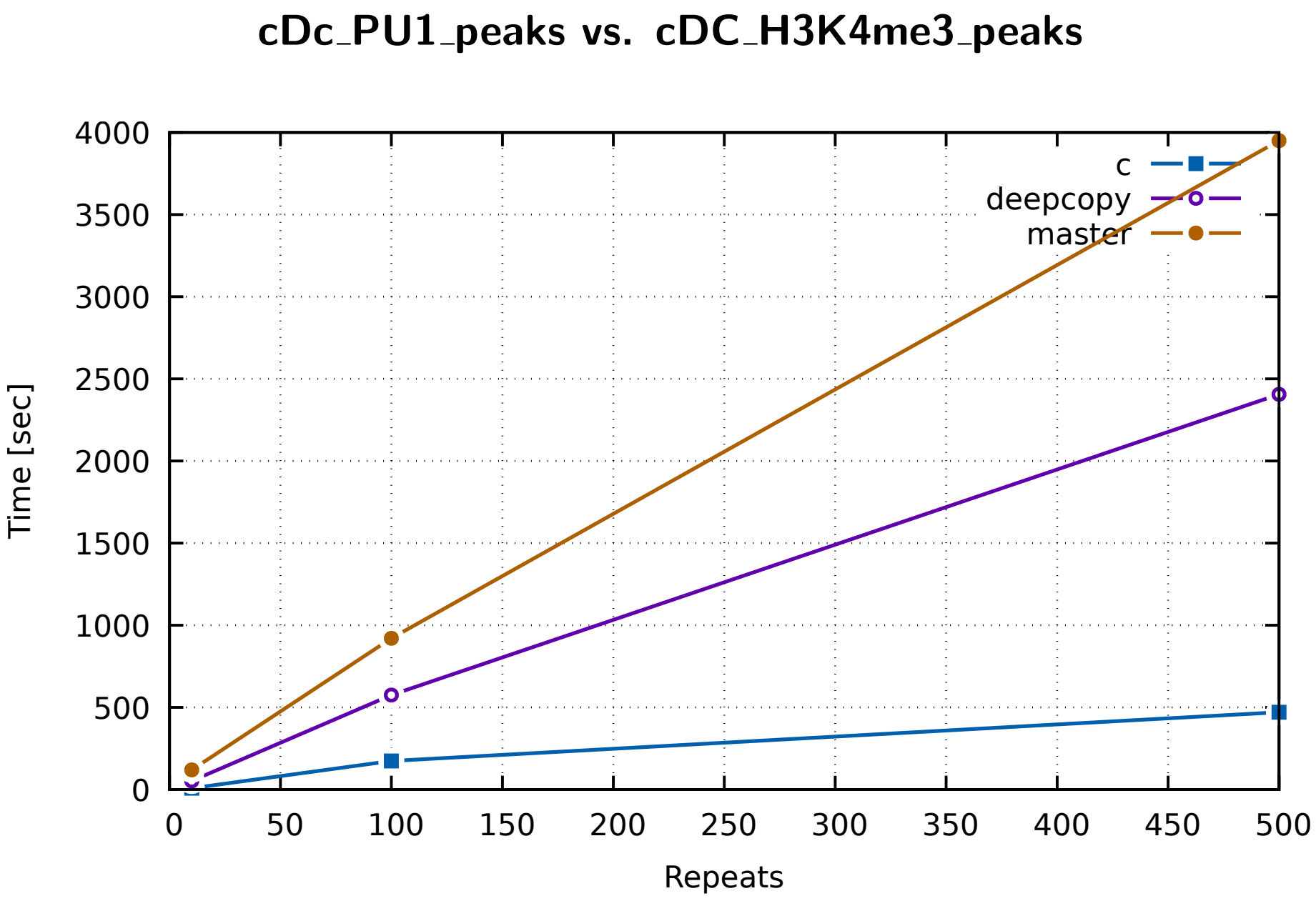
Profiling Jaccard Test after modifications

Ascending order for time percentage ->

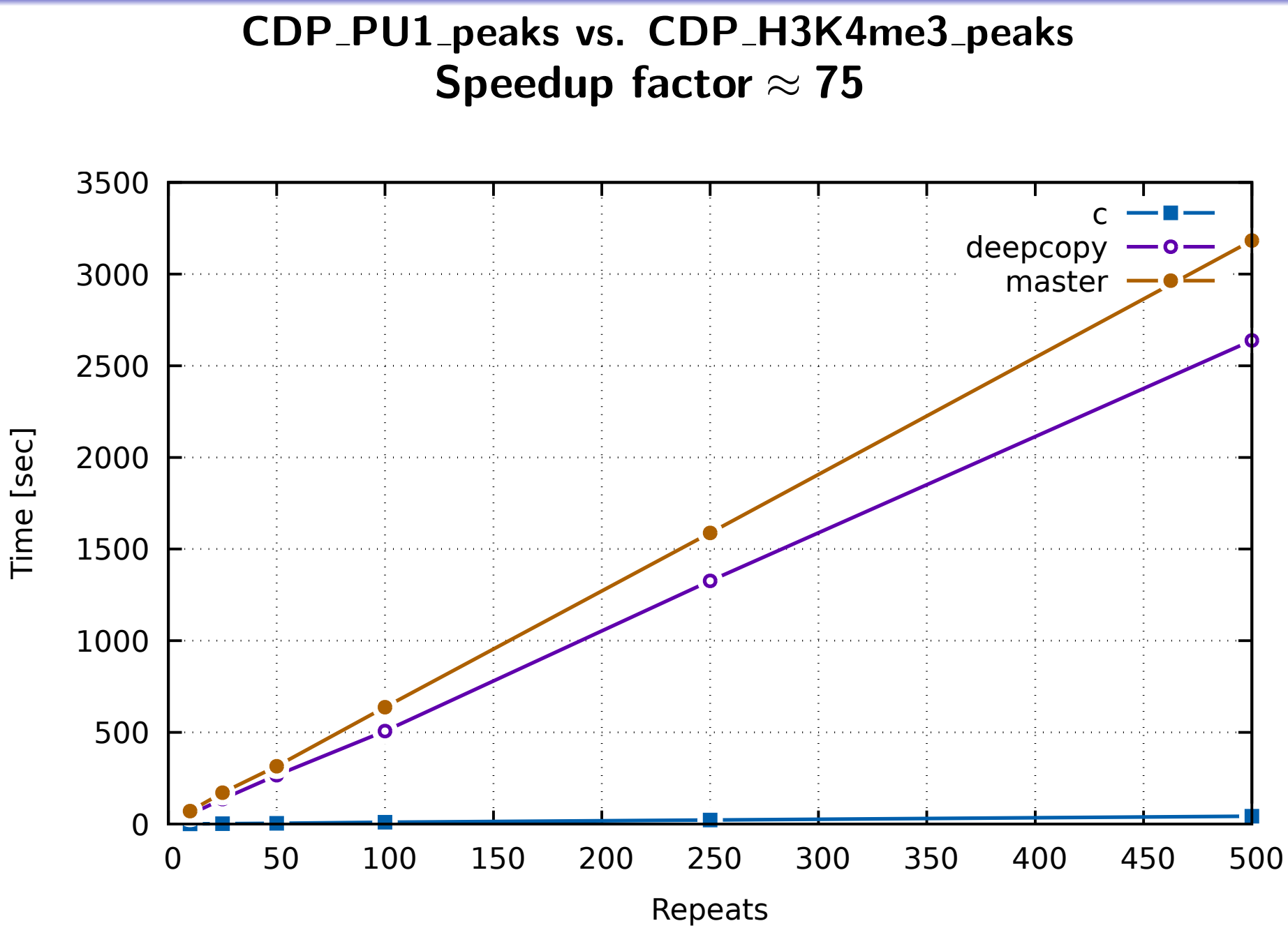
Jaccard Test – Virtual Machine – mm9



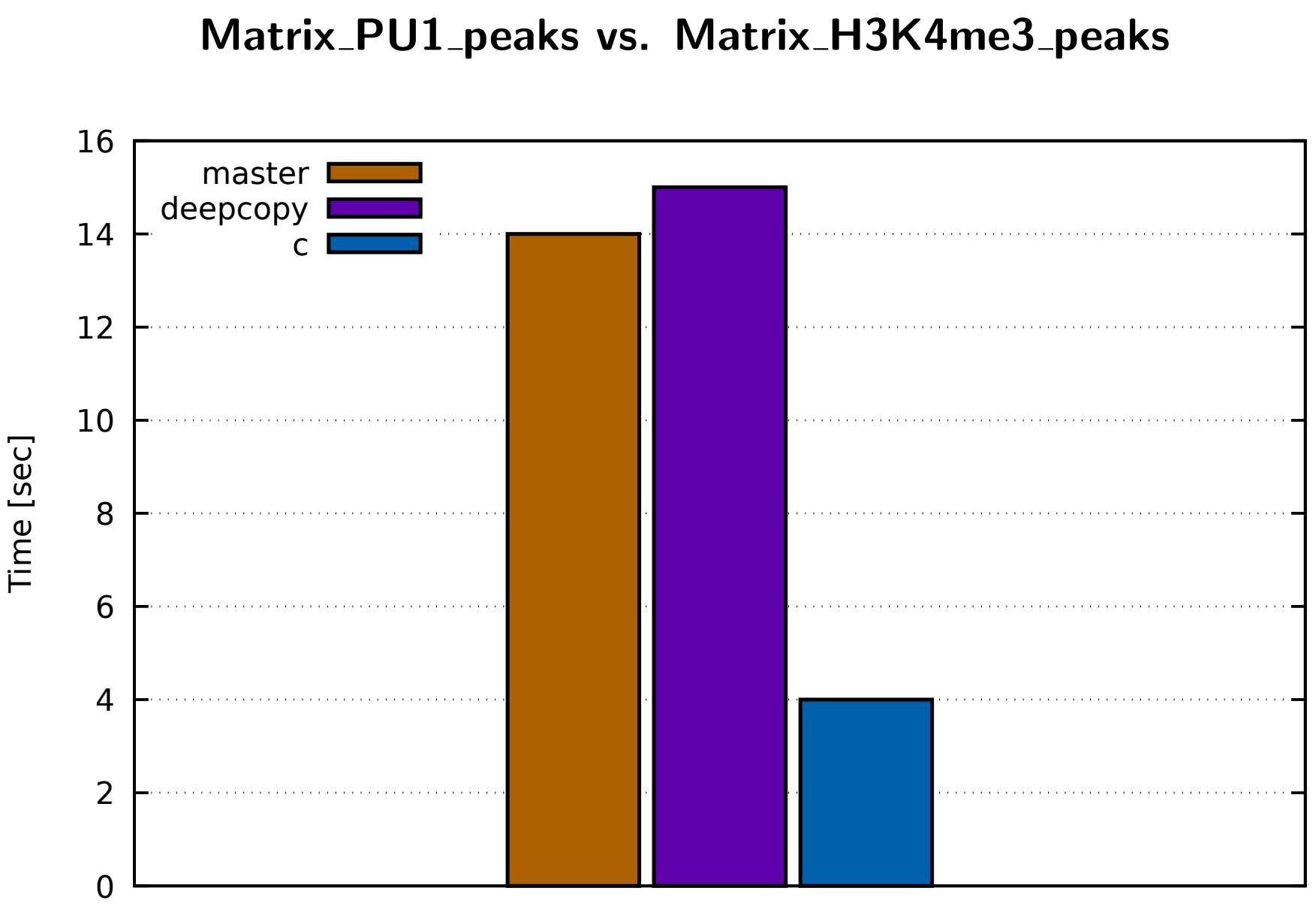
Jaccard Test – Virtual Machine – mm9



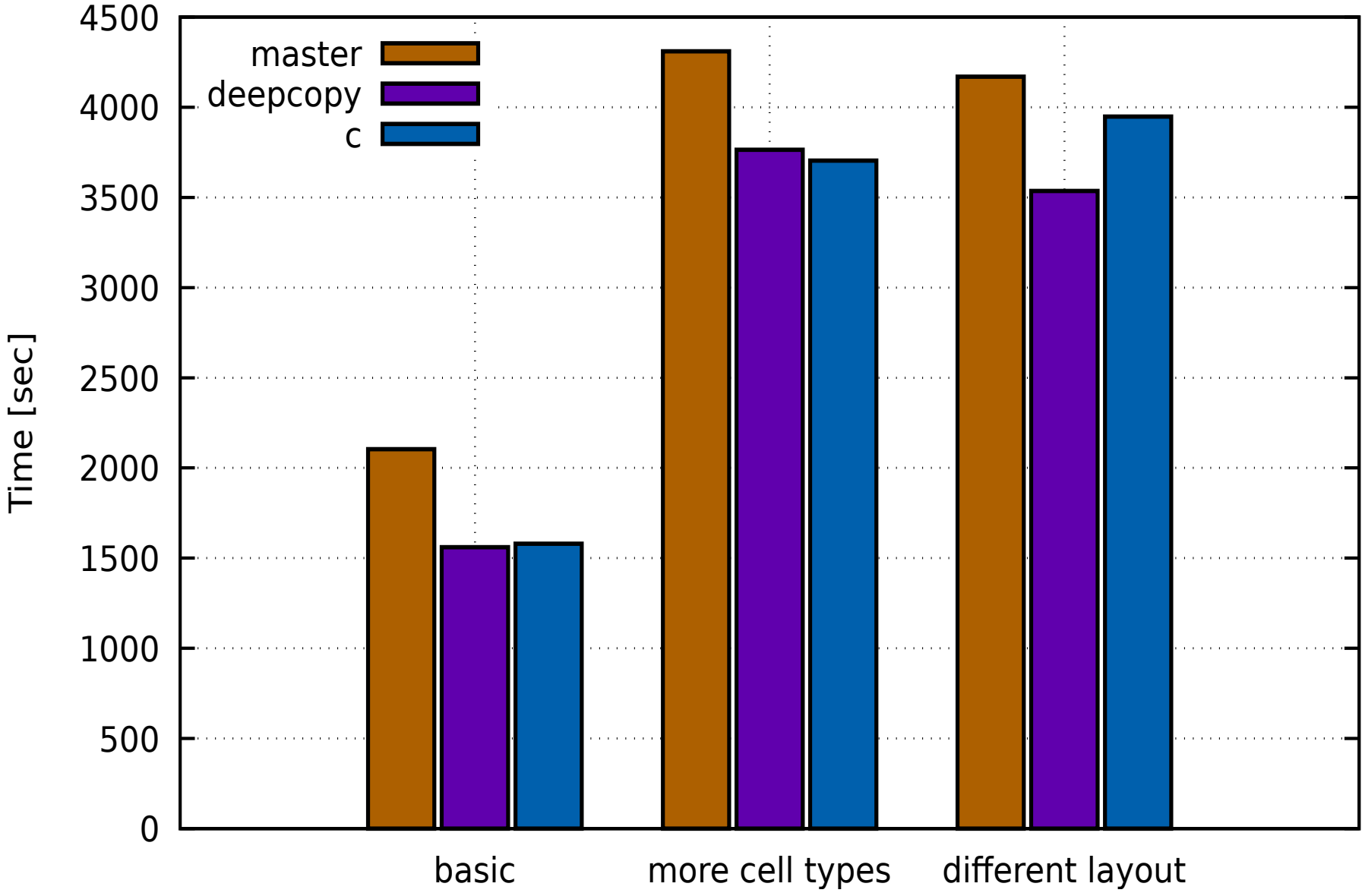
Jaccard Test – Laptop – mm9



Projection Test – Virtual Machine – mm9



Plots – Virtual Machine



The End

Thank you for your attention!