Tim van den Berg, Vincenz Dumann

Calculating the Kullback-Leibler-Divergence on a Genome Scale

Practical Course in High Performance Computing

Table of contents

1 Introduction

2 Sequential Approach

3 Parallel Approach 1

4 Parallel Approach 2

5 Conclusion

Motivation

- Topic
  - Calculate Kullback-Leibler-divergence
  - Working Group of Prof. Beißbarth (medical Bioinformatics)
- Goal: learn how to utilize GWDG resources efficiently

Figure: https://bioinformatics.umg.eu/

Biological Background: Transcription

Disclaimer: Everything is heavily simplified:

- Different cells build different proteins
- Regulated at different stages
  - Here: Transcription of DNA to mRNA
- Regulated by Transcription Factors (TFs) that bind on the DNA
- Goal: find places where TFs bind most likely

Biological Background: Genome

- "Blueprint of the Body"
- Encoded in DNA:
  - 3.1 billion Base Pairs (A, T, C, G)
  - 2m DNA in every cell
  - Organized in 23 (24 for males) different Chromosomes
  - Each Chromosome twice per cell (mom and dad)
  - 3.5 GB text file
  - https://www.ncbi.nlm.nih.gov/genome/guide/human

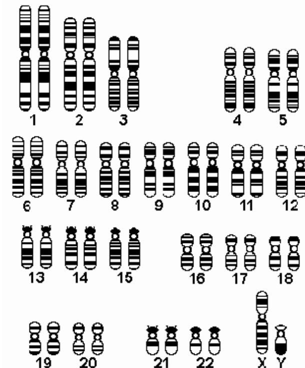


Figure: https://webpath.med.utah.edu

Position Weight Matrices (PWMs)

Position	1	2	3	4	5
A	0.0	0.0	1.0	0.1	0.0
T	0.0	0.3	0.0	0.3	0.0
G	0.5	0.1	0.0	0.5	0.1
C	0.5	0.7	0.0	0.1	0.9

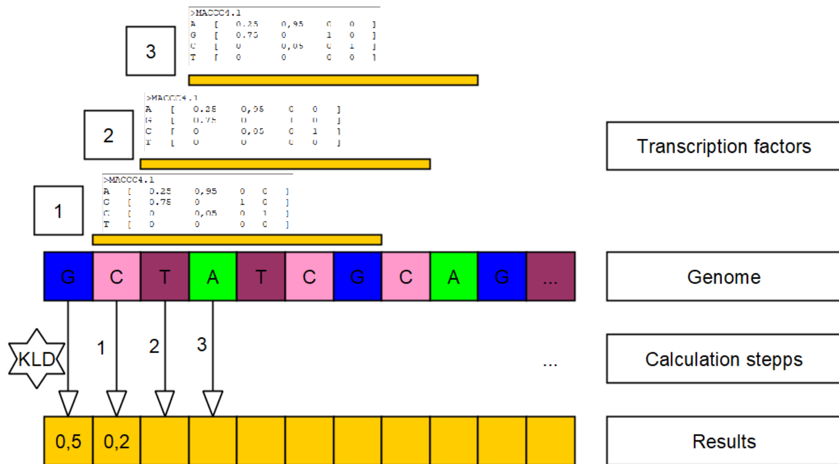
- Transcription Factor PWMs
  - Probability of Base at position in TF binding site
  - 4 Rows (A, T, C, G) and 5-35 columns each
  - total: 1900 PWMs and 24391 Positions
  - 750 kB text file
  - https://jaspar.genereg.net/downloads/

Kullback-Leibler-Divergence

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log(\frac{P(x)}{Q(x)})$$

- Legend:
  - X: A, C, G, T
  - x: single base from X
  - P(x): probability of base (see PWM)
  - Q(x): background probability of x
- gives score for probability of a TF binding site at a position

Algorithm



Technologies used

- Developed in Python
- Used MPI4Py for MPI bindings
- Accelerated using Numba
- Setup via Spack
- Executed via Slurm

Outline

1 Introduction

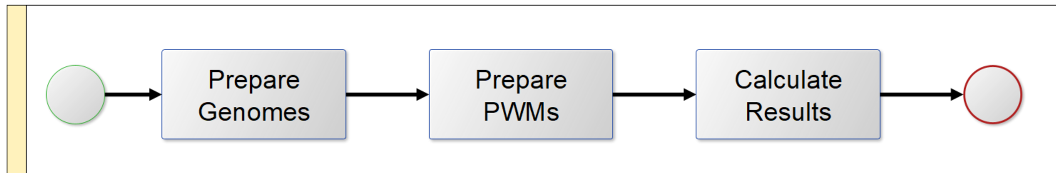
2 Sequential Approach

3 Parallel Approach 1

4 Parallel Approach 2

5 Conclusion

Workflow overview



Core Calculations

```
3  def calculate_kld(sequence: np.ndarray, pwm: np.ndarray,
4                    background: np.ndarray) -> np.ndarray:
5      results = np.zeros(len(sequence) - pwm.shape[1] + 1)
6      for start_idx in range(len(sequence) - pwm.shape[1]): # loop sequence
7          kld = 0.0
8          for cur_pwm_pos in range(pwm.shape[1]): # loop pwm
9              b = sequence[start_idx + cur_pwm_pos] # nucleobase at index
10             if b == 4: # no specific base
11                 # equal to 0, do nothing
12                 continue
13             kld += pwm[b,cur_pwm_pos]*np.log2(pwm[b,cur_pwm_pos]/background[b])
14         results[start_idx] = kld
15     return results
```

$$\mathcal{O}(\text{length(Genome)} \times \text{length(PWM)})$$

Sequential Approach: Measurements

- Whole genome
- One PWM (length 6)
- Local machine (AMD Ryzen 5700, SSD, 16 GB RAM)

Sequential Approach: Measurements

- Local machine (AMD Ryzen 7 5700G, M.2 SSD, 16 GB RAM)
  - $\approx$ 20 GB of RAM needed (started swapping)

Approach 1: Interim conclusion

- $\approx$8 hours for one PWM!
  - Time for 1956 PWMs: $\approx$ 3.7 years
- Python is relatively slow
- Starting point for optimization
- Optimize sequential approach before parallelization

Adding Numba

```python
from numba import jit
@jit(nopython=True, parallel=False)
def calculate_kld(sequence: np.ndarray, pwm: np.ndarray,
                  background: np.ndarray) -> np.ndarray:
    results = np.zeros(len(sequence) - pwm.shape[1] + 1)
    for start_idx in range(len(sequence) - pwm.shape[1]): # loop sequence
        kld = 0.0
        for cur_pwm_pos in range(pwm.shape[1]): # loop pwm
            b = sequence[start_idx + cur_pwm_pos] # nucleobase at index
            if b == 4: # no specific base
                # equal to 0, do nothing
                continue
            kld += pwm[b,cur_pwm_pos]*np.log2(pwm[b,cur_pwm_pos]/background[b])
        results[start_idx] = kld
    return results
```

Sequential Approach: Improvements

- Numba:
  - Compiles decorated function to LLVM
  - Further possibilities with Numba
    - Parallelization
    - Unordered execution
    - GPU utilization (cuda)
- Parallelism is implemented later using MPI

Sequential Approach: Improvements

- Numba:
  - Compiles decorated function to LLVM
  - Further possibilities with Numba
    - Parallelization
    - Unordered execution
    - GPU utilization (cuda)
- Parallelism is implemented later using MPI
- Guesses: How long will it take?
  a. 1 Minute
  b. 10 Minutes
  c. 1 Hour
  d. 5 Hours

Sequential Approach: Measurements

- local machine (AMD Ryzen 7 5700G, M.2 SSD, 16 GB RAM)

Task	Name	Time	Percentage
Wall Clock Time	T_{total}	00:08:09	100.00
Calculations	T_{calc}	00:01:59	23.72
Reading from Disk	T_{read}	00:03:12	39.26
Writing to Disk	T_{write}	00:02:58	36.40
Miscellaneous	T_{misc}	00:00:03	0.61

- $\approx$ 20 GB of RAM needed (started swapping)
- $Speedup = \frac{T_{unopt.}}{T_{opt.}} = \frac{28\ 711s}{489s} = 60.71$
- $\approx$ 14 days for all PWM's

Sequential Approach: Measurements on Server

- single CPU on amp061, no swapping

Task	Name	Time	Percentage
Wall Clock Time	T_{total}	00:14:26	100.00
Calculations	T_{calc}	00:04:07	28.52
Reading from Disk	T_{read}	00:07:02	51.03
Writing to Disk	T_{write}	00:02:55	20.21
Miscellaneous	T_{misc}	00:00:02	0.23

  - 78% slower than consumer PC
  - especially calculating (2 vs. 4 min) and reading (3 vs 7 min)

Sequential Approach: Review

- $\approx$ 8 Minutes for one PWM
  - Time for 1956 PWMs: $\approx$ 14 days
  - Before: $\approx$ 3.7 years
- Great improvement for minimal effort
- Further improvements possible, but we will go parallel now

Outline

1 Introduction

2 Sequential Approach

3 Parallel Approach 1

4 Parallel Approach 2

5 Conclusion

Task- or Data Parallel?

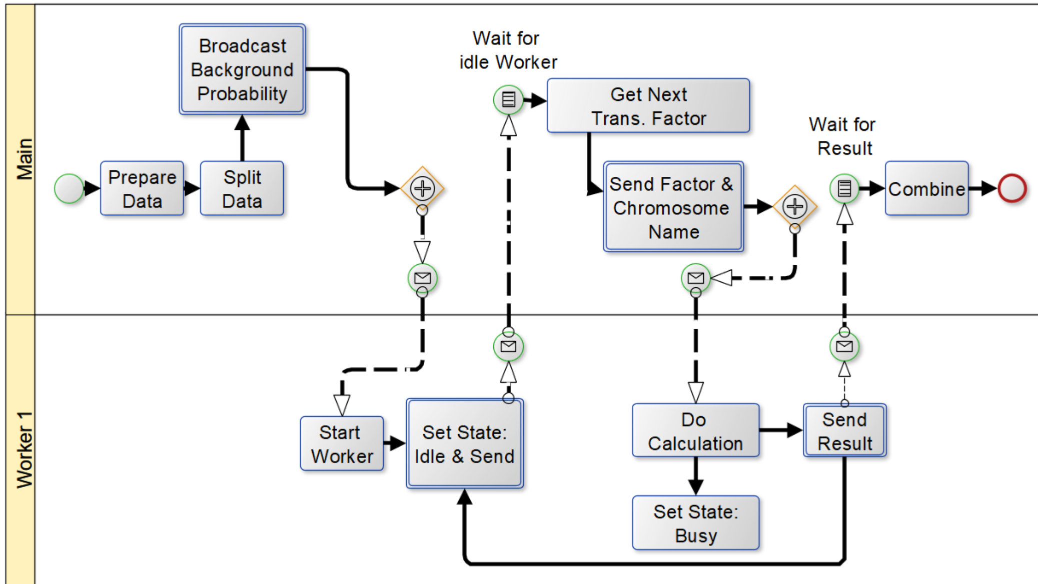
- Task-Parallel
  - Tasks very different working time on all PWMs
  - Tasks depend on each other -> Waiting time possible
- Data-Parallel
  - Calculation steps are independent from each other
  - Possible to split on genome or PWM

Basic Ideas for parallel approaches

- Dynamic amount of workers
- Genome splitted on chromosome level
  - Big chromosomes first
- Scalability: PWM (1956) x Amount of Chromosomes (24) = 46,944 Processes
- Improvement Potential: chromosome split

First parallel Approach: Communication focus

- Focus: Do a lot of MPI
  - Mainly for education
- Main splits the work into packages
  - they are send to workers
  - amount of workers flexible
- Workers calculate and send results back
- Main writes results


Main
Worker 1
Broadcast Background Probability
Prepare Data
Split Data
Wait for idle Worker
Get Next Trans. Factor
Send Factor & Chromosome Name
Wait for Result
Combine
Start Worker
Set State: Idle & Send
Do Calculation
Send Result
Set State: Busy

Parallel Approach 1: Measurements

#Worker	Time	Time/Worker (s)	Speedup	Speedup/Worker
1	39:12	2352	1	1
4	14:05	213	2.75	0.68
8	7:13	54	5.4	0.67

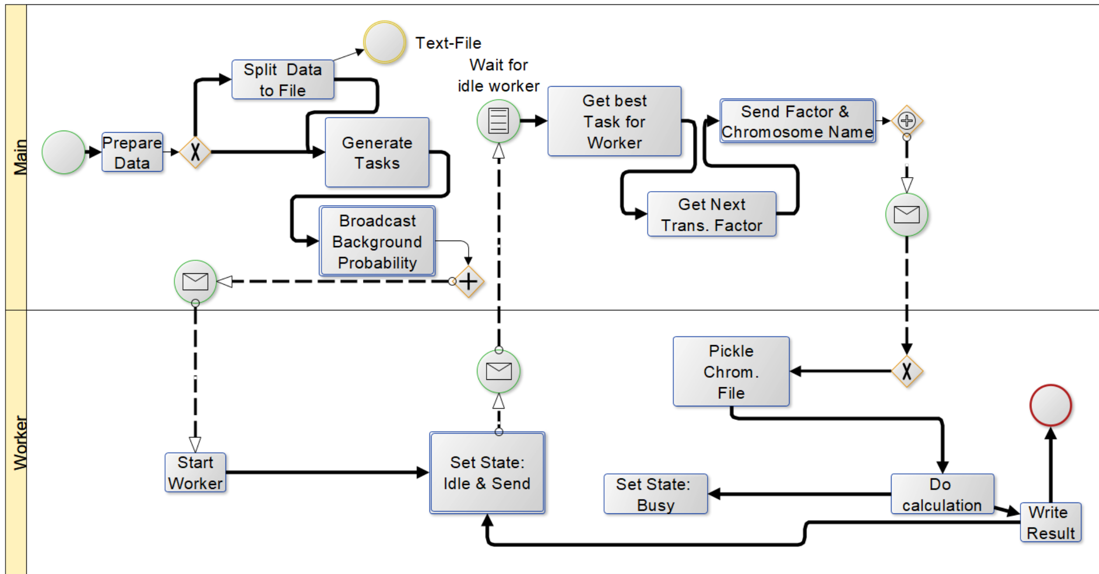
+

Review Parallel Approach 1

- Good Scalability
- High memory consumption!
  - Genome is held in memory by every worker
  - Results are held in memory by main process
- A lot of not needed communication
  - Results are communicated
- Writing the results is a bottleneck in the main process
  - times above are without writing results
  - killed the job after ≈20 min of saving results
  - will be a problem with more than 1 PWM
- this cannot be considered a working solution

Parallel Approach 2: Less communication

- Focus: Performance increase and better memory usage
- Worker reads Chromosome, if needed on its own
- Worker writes result on its own
- Main manages workers

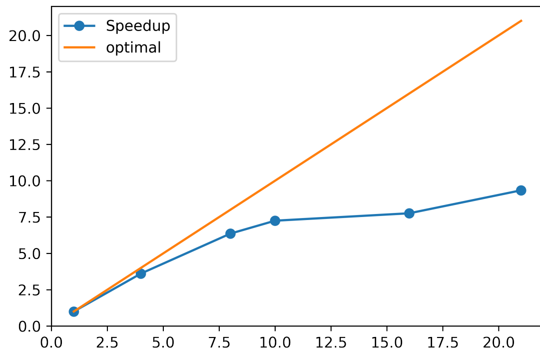

Main
Worker
Text-File
Split Data to File
Wait for idle worker
Prepare Data
Generate Tasks
Broadcast Background Probability
Get best Task for Worker
Send Factor & Chromosome Name
Get Next Trans. Factor
Pickle Chrom. File
Start Worker
Set State: Idle & Send
Set State: Busy
Do calculation
Write Result

Parallel Approach 2: Measurements with 1 PWM

#Worker	Time	Time/Worker (s)	Speedup	Speedup/Worker
1	1:05:15	3915	1	1
4	18:04	271	3.61	0.9
8	10:15	76	6.36	0.79
10	9:00	54	7.25	0.725
16	8:24	31.5	7.76	0.485
21	6:59	19.95	9.34	0.44

Actual speedup vs optimal speedup

- Speedup vs workers
- Speedup limit
  - Data set is to small for 21 workers
  - Limited by longest chromosome
  - No problem with more PWMs



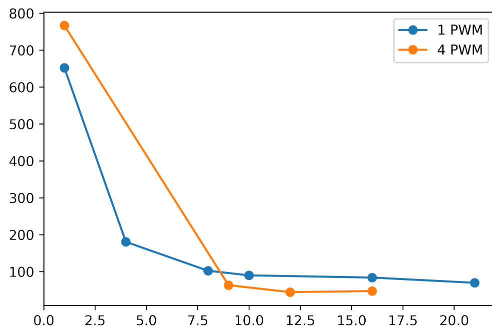
Parallel Approach 2: Measurements with 4 PWMs

4 PWMs are actually 6x longer than 1 PWM

#Worker	len(PWMs)	Time	Time/len(PWM) (s)
1	6	01:05:15	652.50
1	36	06:40:13	767.03
8	6	00:10:15	102.50
9	36	00:38:03	63.42
10	6	00:09:00	90.00
12	36	00:26:45	44.58
16	6	00:08:24	84.00
16	36	00:28:31	47.53
21	6	00:06:59	69.83

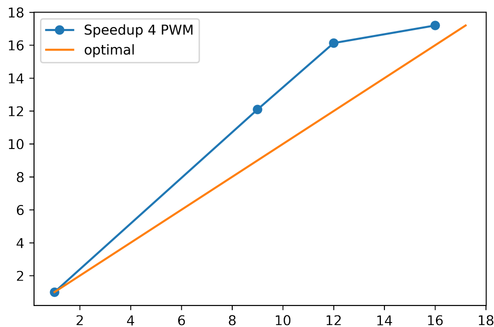
1 PWM vs 4 PWMs

- time/len(PWMs) (s) vs workers
- more efficient with more PWMs
  - reading Genome, etc. shared between workers
  - first datapoint with 4 PWMs probably corrupted



Speedup 4 PWMs

- Unexpected behaviour
  - Better than perfect Speedup
  - One worker took long
  - Only measurement over night
  - Will repeat for report
- needs to be investigated further



Improvement Potentials

- Do more profiling
- Increase granularity
- Use shared memory multiprocessing on single nodes
- Optimize further with Numba
- Use 2bit format
  - Reduces IO and calculation times

Outline

1 Introduction

2 Sequential Approach

3 Parallel Approach 1

4 Parallel Approach 2

5 Conclusion

What we produced

- 4 different solutions
  - Will be returned to Bioinformatic department
  - Can serve as reference project there
  - E-mail us for access to the GitLab

Comments

- Implementation much faster
- Spend a lot of time optimizing IO
  - still not perfect
- Benchmarking was more complicated than expected
  - fell short at the end
- Time management could have been better
- It was a fun project
- Pair programming worked well
  - splitting up could have been faster
  - on the other hand: higher quality?

What you might have learned

- Numba is impressive
  - Python is not a slow language, when used properly
- optimize sequential code first
- problem size matters
- many metrics matter, not only speedup

What you might have learned

- Numba is impressive
  - Python is not a slow language, when used properly
- optimize sequential code first
- problem size matters
- many metrics matter, not only speedup