Parallelization of an R Script for the Automatic Iterative Building of a Hidden Markov Model

Daniele Gader
Department of Computer Science, University of Pisa
Parallel and Distributed Systems course

2nd Semester of Academic Year 2017-2018

1 Application Introdution

1.1 Approach

As part of my Bachelor thesis in Computer Science and Engineering at the Free University of Bolzano-Bozen, I worked on the creation of a method to automatically and iteratively build process models representing the intents of users while interacting with a software system. In such process models, intents are expressed as a set of actions performed by a user to a system to achieve specific use goals [3].
In my method, I applied the theory of Hidden Markov Models (HMMs). HMMs are Markov Chains where the system under study is assumed to have hidden states [8]. HMMs' states represent users' intentions, symbols represent unique user actions and *observations* consist of the overall set of interactions with the system under study.
My approach was inspired by Damevski et al. 's Interactive and Iterative approach [2]: in fact, Damevski et al. build IIHMMs (Interactive Iteratively built Hidden Markov Models), whose construction requires the interaction of a human expert to identify interesting sequences. Instead, my approach is aimed at automatizing Damevski et al.'s approach, as we now build AIHMMs (Automatic Iteratively built HMMs), which require no human intervention for their construction.

1.2 Objective and Dataset

I would like to apply my method for the construction of an AIHMM to the dataset employed by Damevski et al. [1] for the construction of an IIHMM, as decribed in [2] (i.e. carry out a replication study with my method over the same dataset).
Damevski et al.'s dataset contains developers' interactions logs with the Visual Studio IDE.

1.3 Scripts' Functioning Overview

In practice, I created a set of R scripts [4] to mine usage logs of a system for the construction of an AIHMM and IIHMM. In this report, I focus uniquely on AIHMM's construction.
The sequential AIHMM construction procedure is made up of two phases: an *Initialization Phase*, and an *Iterative Phase*.

1.3.1 Initialization Phase

The Initialization Phase represents the program's bottleneck, as it takes more than one month to run with the dataset under study and it is described in detail in Section 3.1, along with a proposed parallelization for it.

1.3.2 Iterative Phase

The iterative phase is aimed at finding an HMM model characterized by the least possible log-likelihood (i.e: the best possibly model). In every iteration, a new state is added, onto that symbols present in the interesting sequences are constrained. The iterative phase is made up of three sub-phases, which are outlined in the following paragraph. If either of the halting conditions outlined in the three sub-phases is met, the iterative phase stops.

1. **Most interesting sequences' identification**: The most interesting sequences are identified based on the dataset's observations and the $\text{HMM}^{(n)}$, where n is the current iteration index. Interesting sequences are defined as sequences that are well captured by the dataset's observations, but are not well captured by the $\text{HMM}^{(n)}$ model [7] . If no interesting sequences are identified, the AIHMM generation phase stops and $\text{HMM}^{(n)}$ is returned.

2. **Log-Likelihood comparison**: $\text{HMM}^{(n+1)}$ is created by adding a new state to $\text{HMM}^{(n)}$. The symbols present in the interesting sequences are then constrained to this new state. Afterwards, the log-likelihood of $\text{HMM}^{(n)}$ and $\text{HMM}^{(n+1)}$ are compared.

$$\begin{cases} \text{HMM}^{(n+1)} \text{ is further checked in bullet (3)} & \text{if log-lik}(\text{HMM}^{(n+1)}) > \text{log-lik}(\text{HMM}^{(n)}) \\ \text{HMM}^{(n)} \text{ is returned and the iterative process terminates} & \text{if log-lik}(\text{HMM}^{(n+1)}) \leq \text{log-lik}(\text{HMM}^{(n)}) \end{cases}$$

3. **Comparison of constrained $\text{HMM}^{(n+1)}$ with unconstrained $\text{HMM}^{(n+1)}$**: In every iteration, after symbols identified in interesting sequences are constrained onto the new state of $\text{HMM}^{(n+1)}$, an unconstrained model of $\text{HMM}^{(n+1)}$ is created and trained (i.e: with no symbols constrained onto the newly added state). Then, the log-likelihood of these HMMs is compared. HMM_C stands for "HMM Constrained", whereas HMM_U stands for "HMM Unconstrained".

$$\begin{cases} {\text{HMM}_C}^{(n+1)} \text{ is passed to iteration } n+1. & \text{if log-lik}({\text{HMM}_C}^{(n+1)}) > \text{log-lik}({\text{HMM}_U}^{(n)}) \\ \text{HMM}_C^{(n+1)} \text{ is returned and the iterative process terminates} & \text{if log-lik}({\text{HMM}_C}^{(n+1)}) \leq \text{log-lik}({\text{HMM}_U}^{(n)}) \end{cases}$$

2 Proposed Parallelization

2.1 Problem Statement

The *initialization phase* of the sequential version of the R scripts for the creation of AIHMMs currently takes more than one month to complete the processing of the whole input dataset under study. On the other hand, the *Iterative Phase* just takes approximately one hour to process every iteration. Ideally, I would like to process this input dataset and future datasets within a few hours with a sufficiently powerful machine.

2.2 Problem solution

Because the computation is a data-parallel one, I would like to decrease the T_c (completion time) of the *initialization phase*, by parallelising functions characterized by a very lengthy latency, which represent bottlenecks in the initialization phase. In the *Initialization Phase*, the bottleneck is represented by functions for the sequences' identification and for the θ - frequent sequences' identification.
Hence, I am going to carry out an analysis of the speedup and scalability achievable in theory and achieved in practice through the analysis of the serial and the potentially parallel part of the program for the *Initialization Phase* of the R scripts.

3 Design - Initialization Phase

3.1 Parallelization of the Initialization Phase

The initialization phase is made up of the following sub-phases. In the following paragraphs, "Workers" is used as a synonym for threads, as the whole analysis was carried out on a shared memory machine.

1. **Dataset loading and pre-processing** A set of M independent datasets of developers' interactions with an IDE (observations) need to be loaded from the disk. One dataset corresponds to actions carried out by one single developer over a few months. Furthermore, some pre-processing of the dataset needs to be performed too, in that outlier symbols are removed.
   Initial Parallelization: In an early parallel implementation, I first merged all the datasets into one single data structure, from which outliers are removed, then I split the datasets onto N available workers, which will then proceed to process them in phase (2). However, this merging and splitting phases actually serialized the program uselessly. The first three stages of Figure 1 represent such operations.
   Final Parallelization: A Map is used for such data-parallel parallelization to reduce the T_c of this phase: datasets are first loaded from the disk in parallel and independently, then these are pre-processed in parallel to remove outlier symbols. Stages "Load Datasets from Disk" and "Remove Outliers" within the Map of Figure 2 represent such operations.

2. **Sequences' identification** Actions in the set of observations loaded from the dataset are grouped into sequences (i.e: actions performed in a timeframe of no more than 30 seconds from one another carried out by the same developer) and a sequence ID is assigned to every sequence. For this reason, datasets cannot be partitioned and need to be processed integrally. (i.e: one dataset cannot be split among two workers unless the temporal ordering of sequences carried out by the same developer is re-established).
Initial Parallelization: A Map is used to reduce the T_c of this phase. The M datasets are assigned statically onto the N workers, which process work as it comes. However, this would lead to severe load-balancing issues in case $M > N$. In fact, datasets are very diverse in size (i.e: the work to carry out is linearly dependent on the size of each file) and the speedup achievable in such parallelization is limited by the largest dataset. Should the largest dataset be scheduled at the end of the computation, this will prolong the computation uselessly. The map of Figure 1 represents such parallelization.
Final Parallelization: An LJF scheduling policy is implemented (Longest Job First) in the map's scheduler on Figure 2. Namely, largest datasets are processed first, to ensure good load balancing in case $M > N$. This is done to avoid that largest files are scheduled in the final phases of the map's computation: this would prolong the computation. Still, the speedup achievable in such parallelization is limited by the largest dataset. After the sequences are marked with the corresponding sequence ID in the map's workers, these are merged, allowing to extract all the sequence IDs from them. Stages "LJF Scheduler" and "Sequences' Identification" of the map in Figure 2 represent such parallelization. The phase (1) and (2) are represented as a map in stage (1),(2) of Figure 3.

3. **HMM initialization and training**: An HMM is initialized with one single state and unique actions in the dataset are taken as symbols. Then, the HMM is trained with the Baum-Welch algorithm, which maximizes symbols' probability in the HMM, based on all observations loaded from all datasets.
Parallelization: No parallelization has been implemented for the Baum-Welch Algorithm, as it is a very complex machine-learning algorithm [6] and does not represent the main bottleneck of the program. In fact, the Baum-Welch Algorithm takes approximately half an hour to to run with all 50 datasets. Stage (3) of Figure 3 represents such phase.

4. **Sequences' Sorting**: After all sequences are identified in phase (2), these are sorted in lexicographical order to allow for efficient θ-frequent and θ-probable sequences' computation, as in [2].
Parallellization: Because sequences characterized by the same sequence ID cannot be split, the resulting partitions of step (2) are passed over to this algorithm and a Map with N workers was meant to be applied to these partitions. However, a Map with N threads proved to cause too many cache faults. Consequently, processing occurs only with 1 thread over the different partitions. Further motivations and analysis underlying this choice are given in Section 5.2. Stage (4) of Figure 3 represents the present phase (4)

5. **θ-frequent sequences identification**: Sequences occurring very frequently in the dataset are identified according to Algorithm 2, as by [2], based on the set of sorted sequences resulting from phase (4).
Parallelization: A map is applied, in that the set of sorted sequences is first partitioned into N equally-sized partitions, where N is the amount of workers available, to ensure good load balancing. Secondly, each partition is assigned to one of the M workers, which applies the sequential algorithm for *theta*-frequent sequences' identification. Finally, the resulting processed partitions are merged. Stage (5) of Figure 3 represents phase (5).

6. **Unconstrained HMM training and new log-likelihood computation**: In this step, an unconstrained HMM with two states is created and trained over the set of observations of the dataset Furthermore, the log-likelihood of this newly created unconstrained HMM is computed..
Parallelization: Analogously to phase (3), no parallelization is applied to the Baum-Welch Algorithm. Stage (6) of Figure 3 represents phase (6).

3.2 Initial vs Final Parallelization for Phases (1) and (2)

Before proceeding with the coding phase of the two proposed parallel solutions, I first analyzed the *Initial Parallelization* and the *Final Parallelization* with the RPLSH tool [5] for phases (1) and (2) of the Design Phase in Chapter 3.1. Following are the resulting designs and the latency of the two parallel solutions.

3.2.1 Initial Parallelization - Phases (1) and (2)

The latency times picked for the different phases corresponds approximately to the order of magnitude of the computations represented.

L = Latency
M = Amount of datasets
N = amount of workers
F_0,…, F_{M-1} = Datasets' on the disk
D_0,…, D_{M-1} = Loaded datasets in random order, with no outliers
D = Single data structure containing all datasets
E_0,…, E_{M-1} = Partitions of sequences

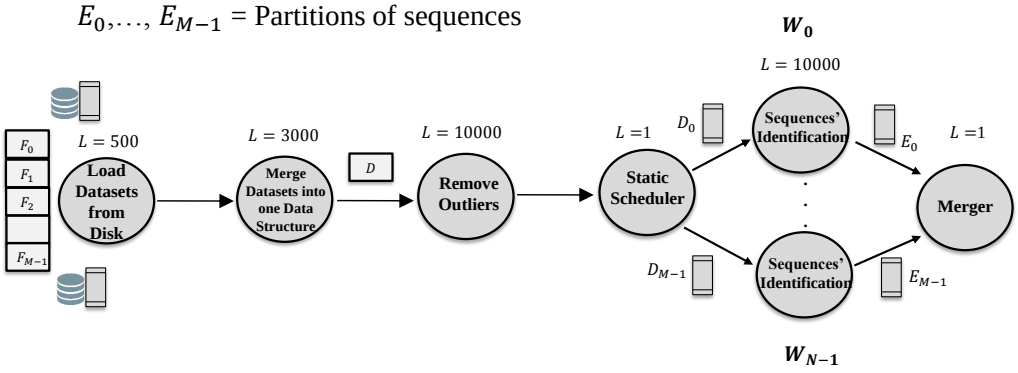


Figure 1: Design of the *Initial Parallelization* for phases (1) and (2)

```
rplsh> load_datasets_from_disk = seq(500, true)
rplsh> merge_datasets_into_one_data_structure = seq(3000, true)
rplsh> remove_outliers = seq(10000, true)
rplsh> sequences_identification = seq(500000, true)
rplsh> sequences_identification_map = map(sequences_identification , 50)
rplsh> initialization_phase = comp(load_datasets_from_disk ,
 merge_datasets_into_one_data_structure ,
  remove_outliers , sequences_identification_map)
rplsh> show initialization_phase by latency
23502.000000     [0] : comp(load_datasets_from_disk ,
comp(merge_datasets_into_one_data_structure
,comp(remove_outliers , sequences_identification_map)))
```

Listing 1: RPLSH code used to design the initial parallelization

3.2.2 Final Parallelization - Phases (1) and (2)

```
FINAL PARALLELIZATION:
rplsh> load_datasets_from_disk = seq(500, true)
rplsh> remove_outliers = seq(10000, true)
rplsh> sequences_identification = seq(500000, true)
rplsh> map_three_steps = comp(load_datasets_from_disk , remove_outliers ,
 sequences_identification)
rplsh> map_three_stages = comp(load_datasets_from_disk , remove_outliers ,
 sequences_identification)
rplsh> initialization_phase = map(map_three_stages , 50)
rplsh> show initialization_phase by latency
10212.000000     [0] : map(map_three_stages) with [ nw: 50]
```

Listing 2: RPLSH code used to design the final parallelization

- Latency for the *Initial Parallelization*: 23502.00
- Latency for the *Final Parallelization*: 10212.00

The latency of phases (1) and (2) is greatly reduced in the *Final Parallelization* of Figure 2 wrt. the *Initial Parallelization* of Figure 1. In fact,the final parallelization's latency is not only reduced because the functions "`load_datasets_from_disk`", "`remove_outliers`", "`sequences_identification`" are executed by the workers of the map, but also because of these facts:

L = Latency
M = Amount of datasets
N = amount of workers
F_0,…, F_{M-1} = Datasets' filenames and sizes
D_0,…, D_{M-1} = Loaded datasets by decreasing size
E_0,…, E_{M-1} = Partitions of sequences

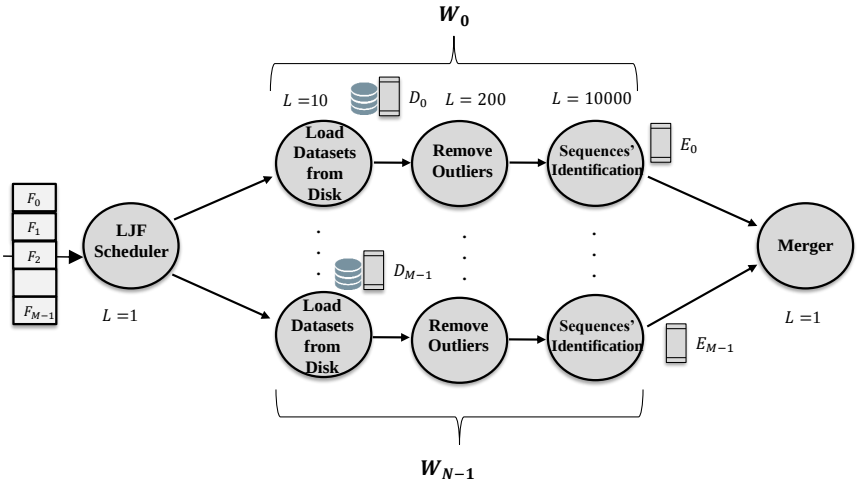


Figure 2: Design of the *Final Parallelization* for phases (1) and (2)

1. Absence of the "`merge_datasets_into_one_data_structure` step of phase (1) in the *Final Parallelization.*
2. Replacement of the Static Scheduler of the *Initial Parallelization* with the LJF Scheduler in the *Final Parallelization.*

3.3 Final Parallelization Outline

Following is a scheme of the full *Final parallelization* scheme used for the program.

M = Amount of datasets
N = amount of workers
F_0,…, F_{M-1} = Datasets' filenames and sizes
D_0,…, D_{M-1} = Loaded datasets by decreasing size
E_0,…, E_{M-1} = Partitions of sequences
G = Sorted sequences
G_0,…, G_{N-1} = Partitions of sorted sequences
H = θ-frequent sequences

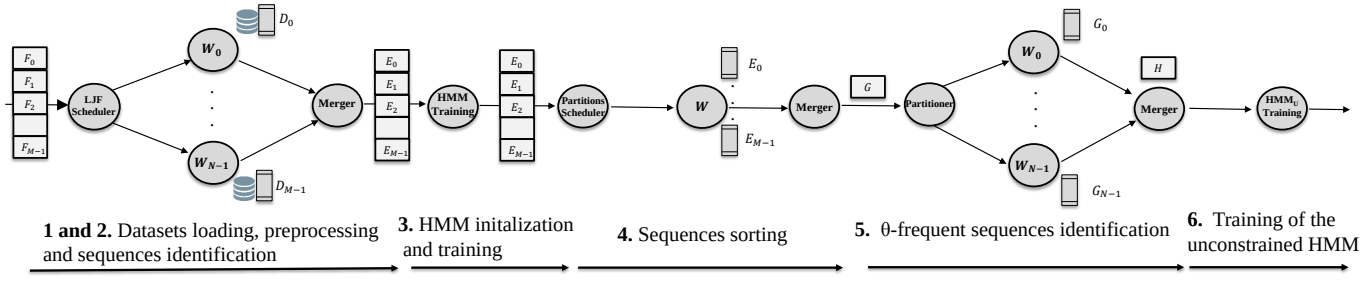


Figure 3: Outline of the *Final Parallelization* for phases (1),(2),(3), (4) and (5) and (6) of the Design Phase in Section 3.1

4 Performance Modelling

Based on the Parallelizations outlined in Figure 3, I modelled the Completion Time of the parallelized initialization phase as follows. The superscript of each step is the index of the phase considered.

$$T_o(N) = T^{1,2}_{Sort} + \mathbf{T^{1,2}_{Datasets}} + T^{1,2}_{Merge} + T^{3}_{HMM} + T^{4}_{SequencesSort} + T^{4}_{Merge} + T^{5}_{Partition} + \mathbf{T^{5}_{\theta-freq}} + T^{5}_{Merge} + T^{6}_{HMM_U}$$

The **bold** terms represent the parts of the program that have been parallelized, whereas all the other terms represent sequential parts. Consequently, I expect my program's speedup to be bounded by the sequential parts and by the constraints imposed by $T^{1,2}_{Datasets}$.
$T^{1,2}_{Sort}, T^{1,2}_{Datasets} T^{1,2}_{Merge}$: Phases (1), (2) of Section 3.1. $T^{1,2}_{Sort}$, although not shown in Figure 3, is a pre-processing phase embedded in the LJF scheduling phase, in that datasets are sorted according to their size in a decreasing order. Datasets' sorting was necessary to allow for LJF scheduling, as the scheduler of the "mcmapply" function used to parallelize the sequences' identification step actually assigns data to the workers based on the order of the datasets within the data structure containing the datasets' sizes.

$$T^{1,2}_{Datasets} = \begin{cases} max(TDataset_1, ..., TDataset_M) \text{ if } N > M \\ \dfrac{\sum_{i=1}^{M} TDataset_i}{N} \text{ if } N \leq M \end{cases}$$

Where M is the amount of datasets used and N is the amount of workers. $TDataset_i$ is the time to load, pre-process and identify the sequences contained in the i-th dataset.
T^{3}_{HMM}: Phase (3) of Section 3.1
$T^{4}_{SequencesSort}$ and T^{4}_{Merge}: Phase (4) of Section 3.1. In T^{4}_{Merge} all the sequences sorted in the $T^{4}_{SequencesSort}$ phase are merged into one data structure.
$T^{5}_{Partition}, T^{5}_{\theta-freq}, T^{5}_{Merge}$. Phase (5) of Section 3.1. In $T^{5}_{Partition}$, the sorted sequences from phase (4) are partitioned into N partitions, where N is the amount of workers available. In T^{5}_{Merge}, the θ-frequent sequences are merged into one data structure.

$$T^{5}_{\theta-freq} = \frac{\sum_{i=1}^{M} T_{\theta-freq} i}{N}$$

$T^6 HMM_U$: Phase (6) of Section 3.1.

5 Implementation

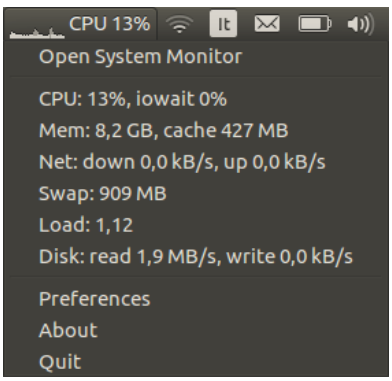
5.1 Implementation Structure

All the code produced and modified for the present project was written in R, an "impure" functional programming language that allows for side effects. Hence, my code is simply a collection of functions that invoke one another. The AIHMM generation code is organized in three main files:

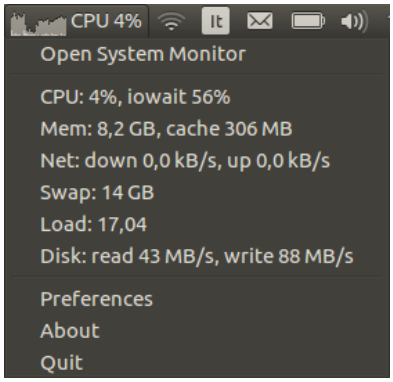
- **AIHMM_Generation.R**: Contains the "main" and invokes functions from the "Common_Functions.R" for the different steps of the AIHMM generation process
- **Common_Functions.R**: Contains all the different functions needed for the AIHMM Generation process. (i.e: steps (3), (4), (5) and (6) of 3.1).
- **Damevski_Preprocessing.R**: Contains functions needed for the pre-processing of the dataset of Damevski et al. [2] (i.e: steps (1) and (2) of 3.1).

All of my code is also available on my Github Repository [1].

[1] www.github.com/DanyEle/HMM



(a) Cache faults while processing sequences with one thread



(b) Cache faults while processing sequenceswith seven threads

Figure 4: Cache faults resulting from running Step (4) of Section 3.1 on my laptop

Sequence ID	Memory Address (HEX)	Memory Address (DEC)
1	6172598	102180248
2	6172538	102180152
3	61724d8	102180056
4	686d4a8	109499560
5	59bf0d8	94105816
...	....	...
17	59bea18	94104088
18	59be988	94103944
19	75958a8	123295912
20	7595848	123295816

Table 1: Memory addresses of the first 20 Sequence IDs of Dataset 235

5.2 Notable Implementation Details

A very notable detail of my program is represented by phase (4) of Section 3.1, namely the Sequences' sorting phase. In the parallel part of this function (namely in the function "generateListsForSequences), firstly all the sequence IDs are extracted by taking the unique Sequence IDs of the data frame produced in Step (1), (2) of Section 3.1. Then, in the following vectorial operation, we iterate over all the unique sequence IDs identified:

```
sequencesLists = lapply(unique(sequences$SequenceID), function(sequenceId)
  sequences[sequences$SequenceID==sequenceId, ]    )
```

Ideally, we would like to parallelize this operation with a map, hence splitting the sequences into N chunks over M workers. However, Table 1 shows that sequence IDs are not always contiguous in memory. This memory discontinuity prevents the program from using spatial locality in caches properly, as we often "jump" to very discontinuous memory addresses and lots of I/O cache faults are generated. When using multiple cores, the amount of cache faults increases even further, as multiple cores content for memory. Figure 4(b) shows the effect of running step (4) of Section 3.1 on my laptop with seven cores: the CPU (an i7 4770HQ) is constantly in an "IOWait" state, whereas the disk is intensely utilized in read and write operations. A similar effect was experienced on the KNL machine, but I had no way to monitor it via the "resource monitor" GUI tool.
On the other hand, by using a single core, less cache faults are generated and less contention for memory is experienced, as by Figure 4(a). Also, no temporal locality (reuse) is exploitable in this computation either, because one sequence ID is discarded after using it.

6 Experimental Validation and Speedup Analysis

6.1 Experimental Methodology

The initialization phase for AIHMM construction was ran with $N = 1, 2, 4, 8, 16, 32, 50, 64$ and 128 workers. The amount of datasets used (M) was 50 due to time constraints in running the experiments. All the experiments

were ran 10 times, and the minimum and maximum of each measurement were discarded. All the times are expresses in seconds (s).

6.2 Serial and Parallel Part, Overall Time

The *Serial Part* $T_s(N)$ uniquely consists of the completion time of the serial parts of the program. The *Effective Parallel Part* $T_p(N)$ consists of the completion time measured for the parallel parts of the program ($T^{1,2}_{Datasets}$ and $T^{5}_{\theta-freq}$)

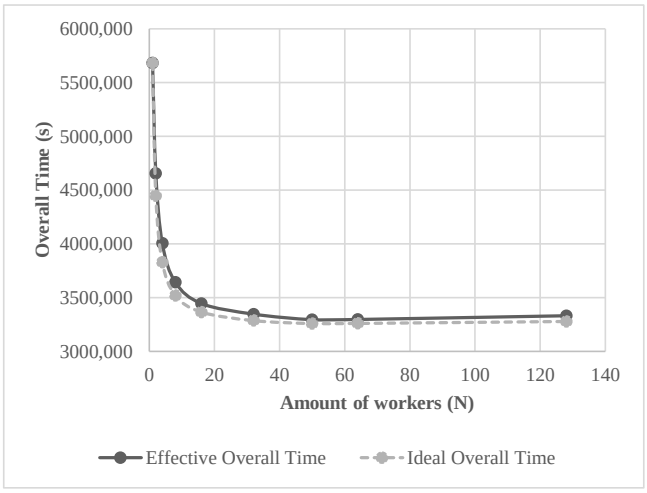
$$T_s(N) = T^{1,2}_{Sort} + T^{1,2}_{Merge} + T^{3}_{HMM} + T^{4}_{SequencesSort} + T^{4}_{Merge} + T^{5}_{Partition} + T^{5}_{Merge} + T^{6}_{HMM_U}$$

The *Ideal Parallel Part*, the *Effective Overall Time* and the *Ideal Overall Time* are defined based on the performance modelling of Chapter 4 respectively as:

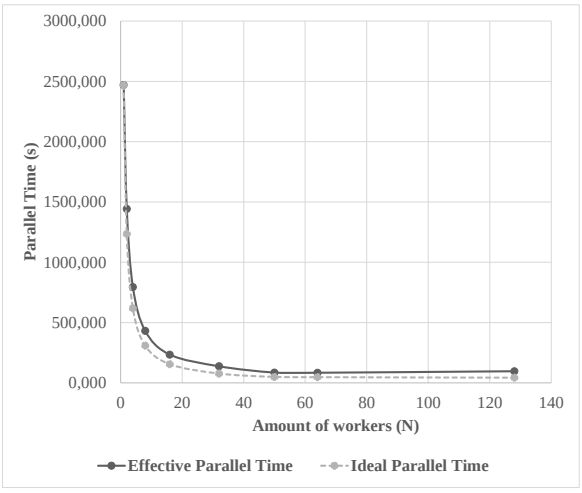
$$T_{IDp}(N) = T^{1,2}_{Datasets} + T^{5}_{\theta-freq} \qquad T_o(N) = T_s(N) + T_p(N) \qquad T_{IDo} = T_{IDp}(N) + T_s(N)$$

Table 2: Measurement of the completion time of the initialization phase ran on the Xeon PHI machine

Amount of Workers (N)	**Serial Part** $T_s(N)$	**Effective Parallel part** $T_p(N)$	**Ideal parallel part** $T_{IDp}(N)$	**Effective Overall time** $T_o(N)$	**Ideal Overall time** $T_{IDo}(N)$
1	3210,764	2470,080	2470,080	5680,845	5680,845
2	3211,661	1443,115	1235,040	4654,776	4446,701
4	3211,032	794,062	617,520	4005,094	3828,552
8	3211,800	430,807	308,760	3642,607	3520,560
16	3211,000	234,184	154,380	3445,183	3365,380
32	3209,106	137,493	77,190	3346,600	3286,296
50	3209,605	85,411	49,402	3295,015	3259,006
64	3213,056	83,873	47,070	3296,929	3260,126
128	3235,353	96,261	42,905	3331,614	3278,258



(a) Plot of the Effective Overall Time $T_o(N)$ and Ideal Overall Time $T_{IDo}(N)$



(b) Plot of the Effective Parallel time $T_p(N)$ and Ideal Parallel Time $T_{IDp}(N)$

6.3 Speedup and Scalability

f, the serial fraction of the program (i.e. the part of the program that does not benefit from improved resources) and p, the parallel fraction of the program (i.e: the part of the program that benefits from improved resources), are defined as:

$$f = \frac{T_s(1)}{T_o(1)} = \frac{3210,764}{5680,845} = 0,565 \qquad p = 1 - f = \frac{T_p(1)}{T_o(1)} = 0,435$$

The *Speedup* $Sp(N)$, namely the improvement in the completion time of the program consequent to the increased usage of resources, is defined as:

$$Sp(N) = \frac{T_{Seq}}{f * T_{Seq} + \frac{(1-f) * T_{Seq}}{N}}$$

T_{Seq} was set to be equal to $T_o(1)$, as $T_o(1)$ represents the best existing sequential version (i.e: the sequential version with least T_c)

$$T_{Seq} = 8022,00\ s \qquad T_o(1) = 5680,845\ s$$

$$\implies T_{Seq} = T_o(1)$$

The *Maximum theoretical speedup* achievable (i.e: with an infinite amount of workers), is:

$$MaxSp(N) = \lim_{N\to\infty} \frac{T_{Seq}}{f * T_{Seq} + \frac{(1-f) * T_{Seq}}{N}} = \frac{\cancel{TSeq}}{f * \cancel{TSeq}} = \frac{1}{f} = \frac{1}{0,565} = 1,769$$

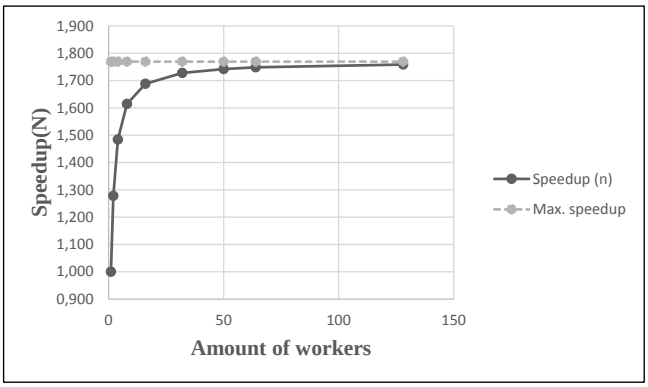
The *Scalability* $Sc(N)$ (i.e: the ability of the program to scale with an increasing amount of workers) and the *Ideal Scalability* $Sc_{ID}(N)$ (i.e: the scalability based on the Ideal Overall time) are defined as:

$$Sc(N) = \frac{T_o(1)}{T_o(N)} \qquad Sc_{ID}(N) = \frac{T_{IDo}(1)}{T_{IDo}(N)}$$

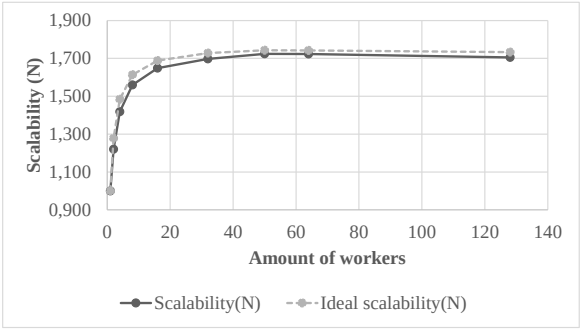
Table 3 shows the experimental results obtained from applying the aforementioned formulas to the measurements of Table 2.

Table 3: Computation of the speedup and scalability based on the measurements of Table 2

Amount of workers (N)	**Speedup** $Sp(N)$	**Scalability** $Sc(N)$	**Ideal scalability** $Sc_{ID}(N)$
1	1,000	1,000	1,000
2	1,278	1,220	1,278
4	1,484	1,418	1,484
8	1,614	1,560	1,614
16	1,688	1,649	1,688
32	1,728	1,697	1,729
50	1,743	1,724	1,743
64	1,748	1,723	1,743
128	1,759	1,705	1,733



(c) Plot of the speedup and the maximum speedup achievable



(d) Plot of the scalability and the ideal scalability

It's worth noting that the maximum scalability is obtained with $N = 50$, where no extra overhead is paid because of time spent in setting up extra idle threads for phase (1) and (2) (i.e: sequences' identification and pre-processing). In case $N > M$, $N - M$ threads would remain idle with no dataset assigned to them. Also, phase (5) (*theta*-frequent sequences computation) can benefit from all the threads being setup up to $N = 64$. Upon increasing the amount of threads beyond 64, the parallel time stops decreasing and even starts increasing, as the overhead in threads' setup is doubled wrt. 64 workers: neither the *theta*-frequent sequences computation nor the sequences' identification parallel phases can benefit from the increased amount of threads.

7 Instructions to run the initialization phase code

1. Log into the Xeon PHI via SSH
2. Adjust the amount of datasets to be used by adding or removing datasets from the folder "/home/spm18-gadler/HMM/Datasets_Damevski_Small", where all datasets are automatically loaded and used by the AIHMM generation code. All existing datasets are stored in the folder "/home/spm18-gadler/HMM/Datasets_Damevski_All".
3. Go to the folder where the R project is located, then run R

   ```
   cd /home/spm18-gadler/HMM    #parallel version
   cd /home/spm18-gadler/HMM_Seq  #sequential version
   R
   ```

4. Load the content of the three files needed :

   ```
   source("Damevski_Preprocessing.R")
   source("AIHMM_Generation.R")
   source("Common_Functions.R")
   ```

5. (Optional) Set an output file

   ```
   sink("file_name.txt")
   ```

6. Run the following command to let the initialization phase begin, passing the amount of workers to be used as a parameter of the function call. [2]

   ```
   run_experiment_workers(N)
   ```

8 Acknowledments

I would like to thank Dr. Massimo Torquati for useful input on how to design the dataset loading and the scheduling policy for phases (1) and (2) in an efficient manner.

References

[1] Kostadin Damevski. ABB Visual Studio Developer Interaction Dataset. https://abb-iss.github.io/DeveloperInteractionLogs/, 2013.

[2] Kostadin Damevski, Hui Chen, David Shepherd, and Lori Pollock. Interactive exploration of developer interaction traces using a hidden markov model. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 126–136, New York, NY, USA, 2016. ACM.

[3] D. Gadler, M. Mairegger, A. Janes, and B. Russo. Mining logs to model the use of a system. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 334–343, Nov 2017.

[4] Russo Barbara Gadler Daniele, Janes Andrea. R Scripts for the creation of AIHMMs and IIHMMs. https://github.com/danyele/HMM, 2017.

[5] Gazzarri and Danelutto. A tool to support fastflow program design, March 2018.

[6] Scientific Software Development Dr. Lin Himmelmann and www.linhi.com. *HMM: HMM - Hidden Markov Models*, 2010. R package version 1.0.

[7] Szymon Jaroszewicz. Using interesting sequences to interactively build hidden markov models. *Data Mining and Knowledge Discovery*, 21(1):186–220, Jul 2010.

[8] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[2] If wanting to run the script on a local machine, further details can be found in the readme.txt file.