CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Parallel construction of variable length Markov models for DNA sequences

Master's thesis in Computer science and engineering

Jan Rune Qvick

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Parallel construction of variable length Markov models for DNA sequences

Jan Rune Qvick

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Parallel construction of variable length Markov models for large DNA sequences

Jan Rune Qvick

© Jan Rune Qvick, 2020.

Supervisor: Alexander Schliep, Department of Computer Science and Engineering
Examiner: Peter Damaschke, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

•

Parallel construction of variable length Markov models for large DNA sequences
Jan Rune Qvick
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern CPUs that contain multiple cores allows for parallel execution of algorithms, and while the technology exists it is not always used by existing implementations. Within this project one such case is investigated, namely the construction of variable length Markov models (VLMC).

This work builds upon the unpublished work of J. Gustafsson, a base implementation for the construction of VLMCs on DNA-sequences. In addition to implementing a parallel variant, the focus has also been on constructing models for large genomes, something not yet undergone within the base project. The report presents two potential practical parallel variants of this base, and early on selects the most promising for further analysis. For this selected approach multiple tests are performed to present runtime, speedup and memory consumption. The load distribution is also analysed, and presents an opportunity for future improvement.

The highest level of speedup was approximately a factor of 7, on 32 cores, compared to seriel execution. This test was performed with an input string of 22 GB. The memory footprint of the implementation, albeit high, is expected because of the adaptation to large input sizes.

Keywords: variable length Markov models, VLMC, parallel computation

Acknowledgements

I would like to thank my supervisor Alexander for insight, guidance and joyful meetings; and my co-supervisor Joel for all his assistance and invaluable discussions. I would also like to thank my family and friends for support, discussion and constructive-critique during my work.

Jan Rune Qvick, Gothenburg, April 2020

Contents

List of Figures

List of Tables

1

Introduction

DNA is a molecule that is found in all living organisms, it contains all the genetic information necessary to code all proteins utilised by that organism. The DNA molecule can be represented by a string called the DNA-sequence. The DNA-sequence is a unique identifier of an organism and its relatives. For this reason research has been carried out in the field of bioinformatics to create computational representations of DNA sequence in order to compare DNA in different ways. One of these approaches is via the use of variable length Markov models (VLMM).

A lot of fascinating work has been carried out with regards to VLMMs. Dalevi et al. [1] identified foreign DNA in the sequences of bacteria, which indicates traces of horizontal gene transfer, a process with which bacteria share genes between individuals rather then inherit them from their respective parent. Borodovsky and McIninch [2] used VLMMs in order to find genes within the DNA sequence. Moreover, this type of model is not exclusively used within the field of bioinformatics. They are present in many other disciplines as well. A few examples are text classification as shown by Ifrim et al. [3], the prediction of likely new targets of cyber-attacks by Fava et al. [5] and even in unexpected areas, such as the research performed by Galata et al. [4], where they analysed human behaviour.

VLMMs are known to be time consuming to train in practice for large sets of data, and for this reason this project focuses on improving the existing performance of the training process with regards to large input data, more specifically large genomes. The work is heavily based on the currently unpublished work of J. Gustafsson and his variable length Markov Chain construction implementation, implemented after that of Shultz et al. [8]. This implementation by J. Gustafsson is referenced as the base implementation throughout this thesis.

The rest of the report is organised in the following manner. Chapter 2, Theory, presents the basis for the understanding of the project as a whole and the necessary knowledge to follow the examples presented throughout the thesis. Chapter 3, Methods, introduces two investigated approaches with regards to the paralellisation and details on how the implementation is verified with respect to accuracy and performance. Chapter 4 is concerned with the results of said evaluation and the last chapter of the report contains a discussion regarding the results obtained and the conclusions of the research.

1.1 Delimitation

This project focuses solely on the optimisation via parallelism, and adaptation of the *PST classifier* project developed by J. Gustafsson, to larger data-sets. Throughout the project, changes to the source repository have been taken into account as long as it was manageable to do so, with respect to the time limitations in place.

2
Theory

2.1 DNA

While it is possible to generate variable length Markov models for any type of sequential input data, in this particular project the focus will be on DNA sequences. DNA is a molecule which carries the genetic information of an organism and the DNA sequence is a way of representing this molecule as a string of characters. These characters are ”A”, ”C”, ”G” and ”T”. Each character represents one of the 4 nucleotides: adenine (A), cytosine (C), guanine (G) and lastly thymine (T). For the purpose of this project the DNA sequence will simply be seen as a long string. The relationships between nucleotides and their function for encoding proteins will not be taken into consideration.

2.2 Variable length Markov models

This section presents the basics of Markov chains, some background regarding variable length Markov models and the data structures used to represent this type of model. Within this section the terminology used follows that used by D. Gusfield [9] and Giegerich et al. [6]. More specifically the extended terminology is presented in table 2.1.

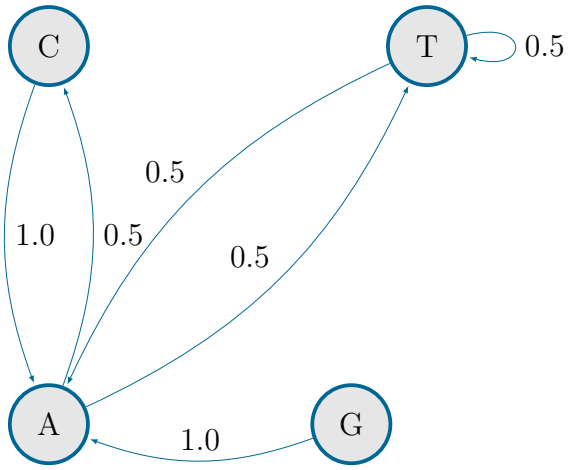
Table 2.1: *Table presenting terminology used.*

Term	Meaning
Σ	The alphabet
$\|\Sigma\|$	Size of the alphabet
t	Minimum number of occurrences for inclusion
K	Pruning argument
k	Order of Markov Chain
$N_s(\sigma)$	Occurrences of sub-string σ

2.2.1 Markov chain

A Markov chain is a type of stochastic model that represents a sequence and is typically used to statistically predict characters based on previous events. Using a string as an example, the Markov chain contains a state for each character of

the alphabet. In addition to the state the model includes probabilistic information about the transition to other states. A graphical representation of the Markov chain constructed from the word "GATTACA" can be seen in figure 2.1. What



Charater	Counts			
	A	C	G	T
A	0	1	0	1
C	1	0	0	0
G	1	0	0	0
T	1	0	0	1

Figure 2.1: *To the left is a graphical representation of the string "GATTACA", and to the right is the tabular representation containing the counts of each pair.*

is described above is a Markov chain of order 1 as only one symbol is taken into account when evaluating the next step. A Markov chain of order 2 would take two symbols into account. Following the example above 'GA' would have a probability of 1 to transition to the character T.
As the order increases, the chain loses generality and becomes more tuned to the given sequence. Additionally, the number of states increases exponentially following $|\Sigma|^k$. Formally, the definition of a first order Markov chain is as follows.

Definition: A first order (discrete-time) Markov chain with finite state space $\mathscr{X}$ is a sequence $X_0X_1X_2...X_n$ of $\mathscr{X}$-valued random variables such that for all states $x_n, x_{n-1}, x_{n-2}, ...$ and all time $n = 0, 1, 2....$

$$P(X_n = x_n | X_n = x_{n-1}, X_{n-2} = x_{n-2}...X_{n-k} = x_{n-k}) \tag{2.1}$$

where for a first order Markov chain the probability $P(x_n|x_{n-1})$ depend only on the states x_n, x_{n-1} and not on the time n or the previous known states $x_{n-2}, x_{n-3}....$ The number $P(x_n|x_{n-1})$ is called the transition probabilities of the chain. For higher order Markov chains the transition probability takes into account k previous states, $x_{n-1}, x_{n-2}...x_{n-k}$.

2.2.2 Variable length Markov chain

Variable length Markov chains (VLMCs) in essence are an extension of the Markov chain concept presented previously. The major difference is that the order k is allowed to vary, rather than being fixed. By letting the order vary the exponential increase of parameters can be limited. This allows the model to be highly tuned to

the input sequence while still keeping the number of states low.

The order k varies depending on the recent history of the sequence, and whether that history is included is decided by taking into account statistical information inferred from the sequence itself. This statistical information is used to exclude or include certain states depending on a threshold, for example the occurrence of the state within the sequence. Many different data structures could be used to represent a VLMC. However, the one used for the base implementation as initially presented by Schulz et al. [8] is the probabilistic suffix tree (PST). PSTs are an extension of traditional suffix trees where each branch has an accompanied vector with probabilities. Suffix trees are presented in more detail in section 2.2.3.

For a more detailed definition of the model see [8].

2.2.3 Suffix trees

A suffix tree is a type of data structure that internally contains all the suffixes of any given input string. More specifically, it is a rooted tree in which one can traverse the tree from the root to any leaf and come back with any suffix present in the input string. While the construction is costly, later operations performed on the structure are comparatively cheap. As a visual example figure 2.2 shows a suffix tree created from our previous example string "GATTACA".

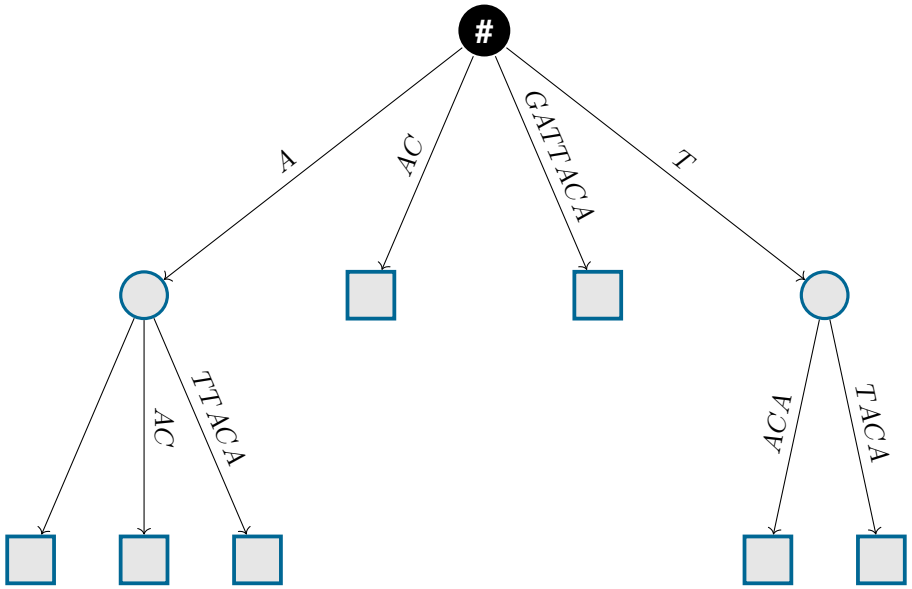


Figure 2.2: *A suffix tree created from the string "GATTACA". The square nodes represent ends of suffixes, while the circular nodes represent branching nodes.*

There are multiple algorithms that generate a suffix tree given an input string, most of them in linear time [10, 11]. The one used in the base implementation is the a lazy suffix tree construction algorithm named *WOTD algorithm* by Giegerich et al. [6]. This implementation enjoys an expected time complexity of $\mathcal{O}(n \log n)$ with a worst case of $\mathcal{O}(n^2)$. Additionally, this algorithm allows us to save memory by not

constructing the full suffix tree. Further information about this algorithm will be presented in subsection 2.2.4.

In order to link together the concept of Markov chains and suffix trees, we can extend the tree presented in 2.2 with the computed probabilities for each branch, and make it so that every node holds a vector that corresponds to the probability that either of its branches could be next in the sequence. This probability vector is calculated by counting the occurrences of each of the branching nodes children's corresponding suffixs, in the input string. More formally, as presented by Kurtz et al. [8]

$$P(\sigma|r) = \frac{N_s(r\sigma)}{N_s(r*)}, \tag{2.2}$$

where $N_s(r\sigma)$ is the number of all, possibly overlapping, occurrences of the subsequence $r\sigma$ in the input string and $N_s(r*) = \sum_{\sigma\in\Sigma} N_s(r\sigma)$. From this information, each of the children is assigned a corresponding probability. With this probability present, the model now contains information unique to the specific input string. Since the probabilities take into account the difference in distribution of the suffixes found in the string, two such strings with different composition, can now be discerned from each other.

2.2.4 Lazy suffx tree construction

The base implementation uses a lazy top down approach suggested by Giegerich et al. [6]. The general idea regarding this approach is the concept that nodes only need to be expanded as they are visited if they satisfy a given criterion. In the base implementation, this is done via a breadth first iteration starting from the root and iterating layer by layer until no more nodes should be expanded. Whether a node should be expanded or not is determined by the statistical parameter t, which is passed to the algorithm as an input argument. Only nodes whose suffix occurs more then t number of times in the input string is included in the tree. This reduces the amount of nodes that will be present in the memory throughout the construction as they will not be pruned away later. Subsequently, it also improves construction time as less nodes need to be visited. Formally, as defined by Kurtz et al. [8], the resulting suffix tree contains all nodes that satisfy

$$T \leftarrow \left(r|r \in \bigcup_{i=o}^{L} \Sigma^i \; and \; N_s(r) \geq t) \right), \tag{2.3}$$

where t is the minimum number of occurrences specified to the algorithm and $N_s(r)$ is all occurrences of suffix r in the sequence. When a node is expanded its parent and some additional flags are placed on two corresponding vectors. The flags contain information whether the nodes have been visited and/or expanded. The table contains a integer pointing to a position in the sequence where the suffix starts.

In addition to the base tree structure, as generated above, a subsequent step adds what is known as suffix links. These suffix links connect branching nodes in a the

tree with their corresponding prefixes further down in the tree if they are present. In essence, it allows for faster traversal of the tree. The suffix links are implemented according to work carried out by M. G. Maaß [7].

Lastly, the table is pruned according to one of two estimators, Kullback–Leibler or Peres-Shield. For this project the estimator Kullback–Leibler is used and it follows

$$\sum_{\sigma \in \Sigma} \left(P(\sigma|ur) * ln \frac{P(\sigma|ur)}{P(\sigma|r)} \right) * N_s(ur) < K, \tag{2.4}$$

where K is given as a parameter to the construction algorithm.

2.2.5 VLMC construction in short

The generation of a VLMC can be presented in short via the following steps:

1. Sequence is read form file
2. List of all suffixes are generated
3. List of all suffixes are traversed and the suffix tree is built according to equation 2.3
4. Suffix links are added
5. Probabilities are calculated
6. The tree is pruned according to section equation 2.4

2.3 VLMC comparison metrics

This section presents fraction shared states and negative log likelihood. Furthermore, the section also describes how they are used and how they are to be interpreted in the context of this project.

2.3.1 Fraction shared states

Fraction shared states (FSS) compute the similarity of two VLMCs by calculating to what extent the two models share nodes. This is done by first constructing a set of all nodes present in both $VLMC_1$ and $VLMC_2$ via the use of intersection. This set, in turn, is used to calculate the proportion of nodes present in each VLMC according to

$$FSS(VLMC_1, VLMC_2) = 1 - \frac{\frac{|shared|}{|VLMC_1|} + \frac{|shared|}{|VLMC_2|}}{2}, \tag{2.5}$$

where each variable represents a set. The *shared* set is the intersection of all nodes. This calculation results in a value between 0 and 1, where 0 represents an exact match.

2.3.2 Negative log likelihood

Negative log likelihood (NLL) is used to measure the probabilistic similarities between two models according to,

$$NLL(VLMC_1, VLMC_2) = \frac{D(VLMC_1, VLMC_2) + D(VLMC_2, VLMC_1)}{2}, \quad (2.6)$$

$$D(x, y) = \log(P(S_x|x)) - \log(P(S_x|y)), \quad (2.7)$$

where $D(x, y)$ is the cross entropy and it is calculated by generating a sequence S_x from model x and using this sequence for the comparison. $\log(P(s|x))$ is the negative log likelihood of s given model x. From the two models two sequences are generated; one for each model. The negative log likelihood is then computed according to equation 2.6.

2.4 Parallelisation

The core problem of parallelising anything is, in essence, the distribution of the work in such a way that the available hardware is optimally utilised. The initial task is to determine which part of the program corresponds to the highest proportion of the runtime. That part of the program is where the most considerable improvement in performance is to be found. This section initially presents a mathematical way to derive theoretical speedup from this proportion. It also gives an overview over different general and language based concepts with regards to parallel development.

2.4.1 Amdahl's law

When comparing different approaches of parallelisation, it is essential to allow for a just comparison between their potential theoretical speedup. One method to do this is to apply Amdahl's law,

$$S_{theoretical} = \frac{1}{(1-p) + \frac{p}{s}}, \quad (2.8)$$

where p is the proportion which benefits from parallelisation and s is the speedup factor which corresponds to the number of cores.

In order to use this relationship, a runtime analysis has to be performed on the existing base implementation. This is done in order to determine two values: the total runtime and that of the parallelisable sections. From this, the proportion of parallelisation, p, can be determined. The potential gains of a parallel solution to a serial implementation are directly related to this proportion. Worthy of note is that this technique does not take into account the overhead that occurs due to the addition of parallelisation itself.

2.4.2 Threads

Threads are a lightweight way of dividing the execution of a program into multiple pieces that can then be run in parallel on different cores in the processor. Cores are sub units of modern processors that independently perform calculations. While it is not necessary to run threads on separate cores, doing so allows for the highest benefit as otherwise they would compete for the cores resources.

While the cores perform their calculations independently of one another they share memory. This means that a shared data structure can be worked by multiple threads. The access of the data needs to be controlled, a matter further presented in the following section.

2.4.3 Synchronisation

When threads work on shared data some problems may arise. One of these, and the main thing taken into account for this project, is called race condition. An example of this is when two threads read the same variable at the same time and both perform calculations on it independently and then both write back to memory. In other words, if the value read by thread1 and thread2 is 3 both threads increment the value to 4 and write this back. The problem is that the value should be 5.

The main way errors of this kind have been taken care of is via the use of mutual-exclusion (mutexes) locks. These synchronisation constructs only allow one thread at a time to access critical sections of the code. While a critical section is accessed by a thread it will keep all other threads waiting until that section is clear.

2.4.4 Parallelisation in used languages

The two different languages used within the project in order to facilitate parallelism are Python and C++. The former in the form of a wrapper around the existing base implementation, later referenced as the distributed approach and presented further in section 3.2. The latter is the language used for the base implementation, and as the shared approach modifies this implementation it subsequently also uses C++. Further information about this approach can be found in section 3.3.
Python, as a language, does not on its own support threads running on multiple processors, because of what is referred to as a the global interpreter lock. Therefore, Python is forced to bypass this lock by spawning new threads into additional, separate, interpreters. There are libraries which allow controlling these new interpreters in an efficient manner. The one used within this project is the standard Python multiprocessing library [13].

C++ on the other hand supports many different frameworks for parallelism. For this project the C++11 thread library [14] was chosen as the main library, as it is low enough level be fully controllable via mutexes, while still being high enough to facilitate function calls without the creation of structs for argument data.

3
Methods

3.1 Runtime analysis

In order to assert the potential for improvement over the base implementation, the runtime for each major step has to be measured . This is done using timestamps within the code itself using the C++ chrono library [15]. The step that takes up the highest portion of the program is the initial focus point of the parallelisation effort, followed by the second and so on. The analysis is performed on a subsection of the human genome.

Section 4.1 contains the result of the runtime analysis and indicates that the initial focus should be placed on the construction of the PST and secondly on the addition of suffix links both within the WOTD-algorithm.

3.2 Distributed model

The distributed PST approach consists of four steps. The first step is to split the DNA sequence given as input into multiple sub sequences. These, in step two, are then used as input for the construction algorithm in order to generate separate sub PSTs. Each of these constructions run individually on their own core. The third step is to combine these sub PSTs into a single data structure. The final step is to prune the combined tree as it would have been done in the base implementation.

Because of the split of the data into sub sequences there is a risk that important information will pass unnoticed in the sections where the split occurs. As a counter measure an additional n nucleotides on each side of the split are concatenated to their respective sub set. This will later be referred to as overhead, figure 3.1 shows a graphical schematic representation of overhead with respect to 4 threads.

Since sub PSTs will be generated on a subset of the data, the statistical requirements that dictate whether a node should be present in the tree had to be modified. The parameter that primarily affects this is the min-count argument, as this results in a lower threshold for inclusion. However, a too low min-count will result in a combined PST that contains a high number of nodes that would not be present in the standard approach. Step four, additional pruning, is an attempt to mitigate this fact.

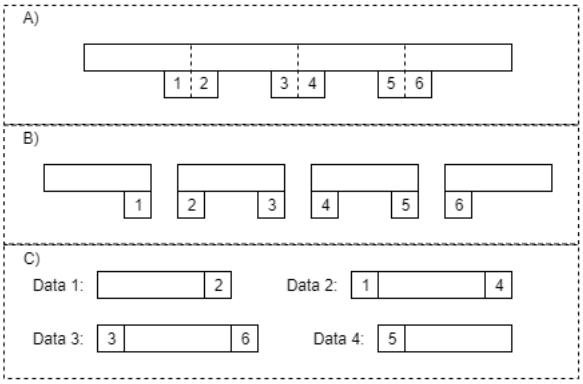


Figure 3.1: *Schematic representation showing overhead for 4 threads. Sub-image C shows the individual sub-sequences feed into each thread and how the overhead of the sub-sequence relates to the initial DNA sequence.*

Another possibility to control the inclusion of nodes within the tree structure is the threshold parameter. This is used within the base algorithm to control how aggressive the pruning of the PST should be. By lowering this value more node will be included when the sub PSTs are being generated.

The challenge is to find a composition of these arguments; overhead, min-count and threshold that allows for the generation of a combined PST that closely resembles that of the base implementation.

3.3 Shared model

The base implementation, as described in subsection 2.2.4, is a lazy top-down approach to constructing a PST. In essence, it is a breadth first iterative approach that starts from the root and constructs nodes as they are visited. On a node visit the algorithm evaluates whether it should be included or not.

The fundamental issue when attempting to parallelise this top-down approach with a shared data structure is that there is no simple way to distribute the workload. Nodes are only expanded when they are visited, and in order for them to be visited, their parent must have been visited already. This top-down relationship means that there is few ways to approach the parallelisation of this construction with a shared memory model.

The key lies in the top-down approach itself as well as some insight about the input data. As the given alphabet that the input data consists of is very short (only four characters), one can expect that the first layer of the tree will always be present. In other words, there will always be a suffix starting with each of the nucleotides in the DNA sequence.

By serially expanding nodes and filling layers down to depth d it is possible to use

these newly constructed nodes as new expansion points. By selecting one of these nodes, n, and calling on the construction algorithm again, but this time n acting as root, this subsequent iteration will generate a subtree containing all suffixes that have n as an ancestor. See figure 3.2 for a visual representation of the concept.

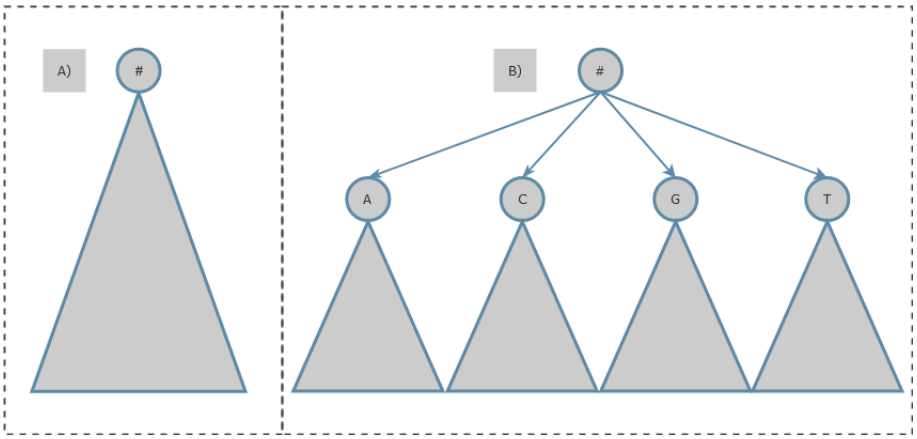


Figure 3.2: *To the left, marked a), represents the serial construction, while the right, marked b), represent the parallel. The first layer of nodes in the parallel construction are done in series after which, depending on initialised layers, 4^d threads are started to continue the construction of their respective subtree. Here d corresponds to 1 which is the depth of the tree where parallel construction is started.*

By doing this for each node on the layer the same layer, d, and performing the subsequent construction on separate processors, there is a potential parallelisation factor of $|\Sigma|^d$, where $|\Sigma|$ is the size of the alphabet, assuming an even distribution of Σ in the input.

In our case, with the assumption that the distribution of nucleotides in a given DNA sequence is roughly equal, the speedup for this improved construction is approximately 4^d. In reality however, this ideal case is not realistically found in biological structures. Therefore, some additional load-balancing is necessary. These measures are not taken into account within the scope of this project. The effect of the lack of load balancing is evaluated and presented in the results.

In addition to the base construction the step where suffix links are added consists partially of breath first traversal which can intuitively be parallelised in the same manner as the construction.

The runtime analysis of the base program, presented in section 4.1, shows that the majority of the runtime is concentrated within the construction of the suffix tree and the addition of suffix links. Together, these steps correspond to roughly 90% of the runtime and from this information the plotted theoretical speedup, calculated via Amdahl's law, is shown in figure 3.3.

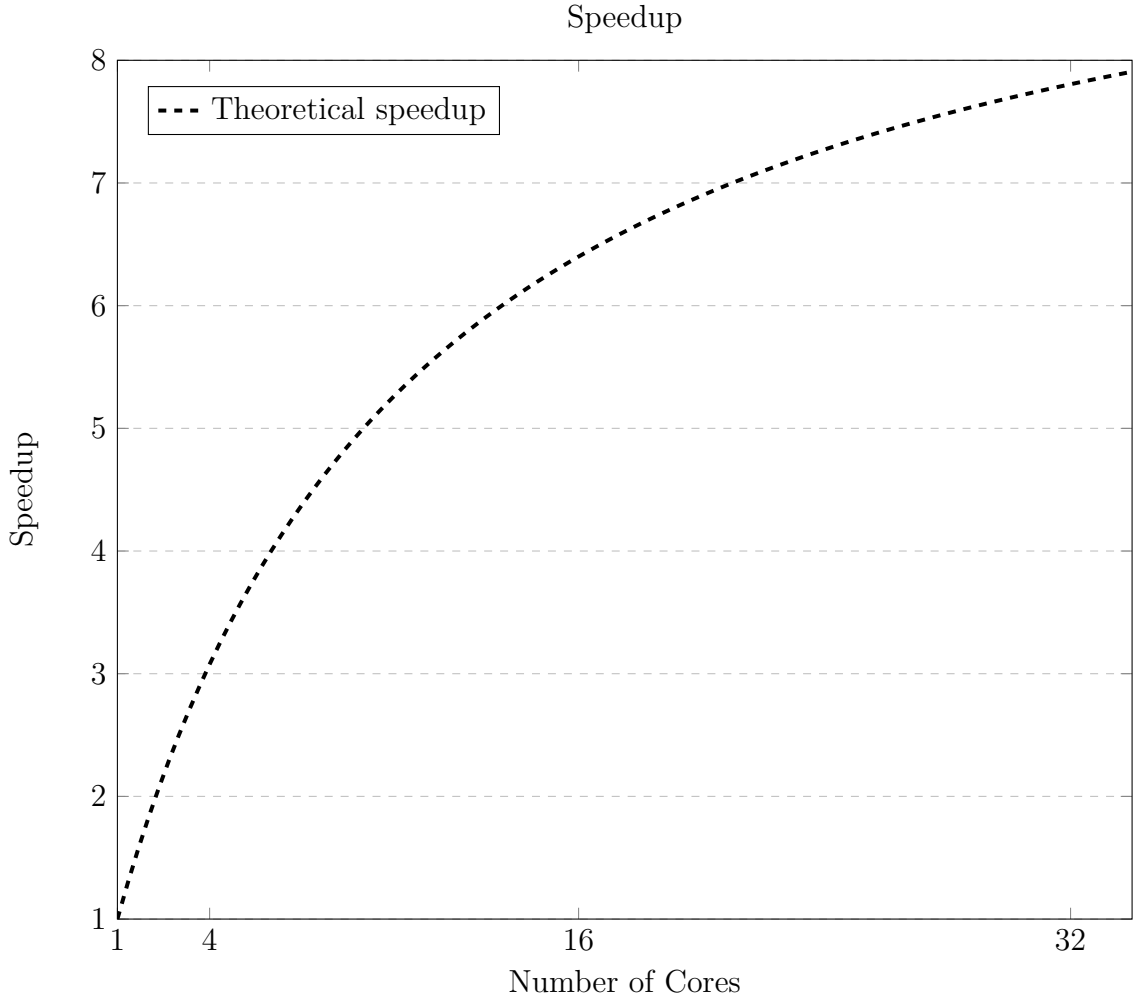


Figure 3.3: *Graph presenting the theoretical speedup according Amdahl's law for* $p = 0.90$.

3.4 Verification and evaluation

This section presents how the verification and evaluation are performed in further detail. The verification subsection focuses on the confirmation that the generated trees share structure and probabilistic properties. The evaluation subsection is focused on the performance, more specifically, runtime and memory consumption of the parallel model.

3.4.1 Verification

The verification is performed using both the base and improved algorithms to generate individual PSTs for a specific genome. The generated models are then compared with focus on two different metrics. The general structural differences are determined via fraction shared states, and the probabilistic accuracy is determined via negative log likelihood, both further described in Section 2.4.1. For the NLL metric,

a sequence corresponding to 10% of the DNA sequence used for the calculation.

The verification of accuracy is ultimately performed on relatively short DNA sequences as the base implementation cannot construct models for large genomes. The genomes which are utilised for accuracy verification are presented in Table 3.1.

3.4.2 Evaluation of performance

The evaluation of the performance is carried out similarly as the accuracy verification. However, it focuses solely on the implementation developed during this project and does not take the base implementation into account. The duration and memory consumption of VLMC construction using a single processor are compared to that of a few different multi-processor executions. The genomes presented in Table 3.2 are the input for the construction.

3.5 Dataset

The dataset of genomes used to evaluate the accuracy and the performance are presented in Table 3.1 and 3.2. The genomes were gathered from the National Center for Biotechnology Information (NCBI). The genomes themselves are contained in a FASTA files, which are a standardised format for storing genetic information. These files contain multiple records for different distinct areas of the genome. Before being used as input for the construction algorithms, the records need to be merged in such a way that each FASTA-file contains one record with all the genetic information belonging to that specific organism. This is done in order to facilitate comparison between the base and improved algorithm, as the base implementation only loads the first record from a FASTA-file.

3.5.1 Verification of accuracy

The genomes selected for accuracy verification presented in table 3.1 were chosen with two objectives in mind. The first being the value of having all major domains represented in order to verify that DNA sequences that belong to each yield accurate models. Secondly, they were chosen based on their relatively short length as they had to be able to run on both models.

3.5.2 Evaluation of performance

In order to evaluate the performance of the models the dataset presented in table 3.2 is used. These specific genomes were selected because they have some of the longest DNA sequences within their respective domain. The runtime will include the time it takes to read the data from disk, but exclude the time required to write the tree to console output or disk. Min-count $t = 10000$ is used to get results slightly faster, the effect of this is that only suffixes that occur more then 10,000 times are

Table 3.1: *Genomes selected for the evaluation of the performance following parallelisation. All sources are from https://www.ncbi.nlm.nih.gov.*

Organism	Domains	Scientific name	Sequence Length	GenBank identifie
Long-grained rice	Plantae	Oryza sativa	387,424,359	GCA_001623365.2
Fruit fly	Animalia	Drosophila melanogast1	133,403,897	GCA_004798055.1
Thale cress	Plantae	Arabidopsis thaliana	119,668,634	GCF_000001735.4
Baker's yeast	Fungi	Saccharomyces cerevisiae	12,157,105	GCF_000146045.2
E.Coli	Bacteria	Escherichia coli	5,129,890	GCF_008632555.1
Human alphaherpesvirus 3	Virus	Varicella-zoster	124,884	GCF_000858285.1
Ebola	Virus	Zaire ebolavirus	18,959	GCF_000848505.1

included in the final tree. The parameter K is set to the standard 1.2. Peak memory consumption and CPU utilisation will be determined via the Linux included script **/user/bin/time**. All performance tests will be carried out on a system running **Ubuntu 18.04.4 LTS** and kernel version **4.15.0-54-generic**. The system uses a Intel(R) Xeon(R) 6126 and 755 GB of DDR4 memory. More information regarding the system is summarised in appendix A.

Table 3.2: *Genomes selected for the evaluation of the performance following parallelisation. All sources are from https://www.ncbi.nlm.nih.gov/.*

Organism	Domain	Scientific name	Sequence Length	GenBank identifier
Loblolly Pine	Plantae	Pinus taeda	12,342,093,815	GCA_000404065.3
Larix sibirica	Plantae	Larix sibirica	12,342,093,815	GCA_004151065.1
Palaemon	Animalia	Palaemon carinicauda	6,699,723,695	GCA_004011675.1
Human	Animalia	Homo sapiens	3,099,706,40	CA_000001405.28

3.5.3 Evaluation of load balancing issues

This evaluation is only performed on the shared memory model, as the data that goes into each of the generated sub trees can be of different sizes. Since load balancing measures are missing the system resources can not be fully utilised. The impact of this is therefore measured during the construction of the model, via time probing.

4
Results

4.1 Runtime analysis

The results of the runtime analysis are presented in Table 4.1. They indicate that the construction of the suffix tree, within the WOTD-algorithm, is responsible for roughly 71% of the runtime and is therefore the focus of the shared model. Because of Amdahl's law, only parallelising this portion of the code does not yield satisfactory results. Therefore, the addition of the suffix links to the structure, at about 18%, should also be parallelised in order to allow for a higher potential improvement.

Table 4.1: *Runtime analysis performed on a subsection of the human genome NCBI: GRCh38.p13*

Step	Sub-step	Runtime of sub-step (ms)	Proportion in sub-step (%)	Proportion of total (%)
Support Pruning	Build Tree	129912	79.59	71.54
	Expand implicit nodes	140	0.09	0.08
	Add Implicit Status	32685	0.3	0.27
	Add Suffix Links	32685	20.02	18.0
Similarity Pruning	Reverse Suffix Links	8866	48.27	4.88
	Calculate Probabilities	649	3.53	0.36
	Pruning	8852	48.19	4.87

4.2 Distributed model

During initial testing of the distributed model some potentially good results were gathered. However, as the input size grew the quality of the results degraded. This is further described and expanded upon the in the discussion, section 5.1.1.

4.3 Shared model

The following sections present the results of the Shared model. It first shows the verification before looking into the runtime evaluation and load balancing.

4.3.1 Verification

The verification of the shared model, carried out as described in the methods chapter, yields the results presented in Table 4.2. The FSS score indicates that the resulting

structure is the same for most DNA sequences, or in the case of long-grained rice negligibly close. The NLL is the same across the board. This suggests, that while the the exact structure might differ somewhat for longer sequences the probabilities within in the models are strikingly similar, at least when generating test data at the size of 0.1 sequence length.

Table 4.2: *Results of the verification. A serial and a parallel tree are generated for the listed organisms, the distance between them is calculated and noted down. '-' corresponds to time measured below 1 seconds.*
1 and 2 corresponds to Fraction Shared States and Negative Log Likelihood respectively.

Organism	Sequence Length (Base Pairs)	Number of Processors	PST Constrcution (Seconds)	Total Runtime (Seconds)	FSS[1]	NLL[2]
Long-grained rice	387,424,359	1	190	308	$2.32E-07$	0.0
		4	76	188		
Fruit fly	133,403,897	1	66	109	0.0	0.0
		4	26	69		
Thale cress	119,668,634	1	9	20	0.0	0.0
		4	4	15		
Baker's yeast	12,157,105	1	2	4	0.0	0.0
		4	1	3		
Escherichia coli	5,129,890	1	1	1	0.0	0.0
		4	-	1		
Alphaherpesvirus 3	124,884	1	-	-	0.0	0.0
		4	-	-		
Zaire ebolavirus	18,959	1	-	-	0.0	0.0
		4	-	-		

4.3.2 Runtime evaluation

This section presents the runtime, speedup and memory consumption of the Shared model. The evaluation is carried out as presented previously in section 3.5.2. The runtime can be seen in figure 4.1. The one core results seem to indicate that for relatively small genomes, Larix Siberia and shorter, the doubling in input size results in a doubling of runtime. However, when comparing the two larger genomes, Larix Siberia and Loblolly Pine, this relationship is no longer present. Rather, with doubling of input size the runtime is approximately 4 times longer. This observation seems to support the time complexity presented by Giegrech et al. where the linear $\mathcal{O}(n \log n)$ is expected for smaller input data, and as the input size grows the runtime approaches the worse case $\mathcal{O}(n^2)$.

The runtime data is the basis for the speedup calculations, presented in the graph seen in figure 4.2. Here the speedup for the four largest genomes is superimposed on the previously presented graph showing the theoretical speedup seen in figure 3.3. The data indicates that the larger genomes can utilise the additional cores better. Measurements of CPU utilisation can be seen in appendix B.
The memory consumption is almost the same, regardless of the number of cores used, see figure 4.3. From this information the average peak memory consumption per

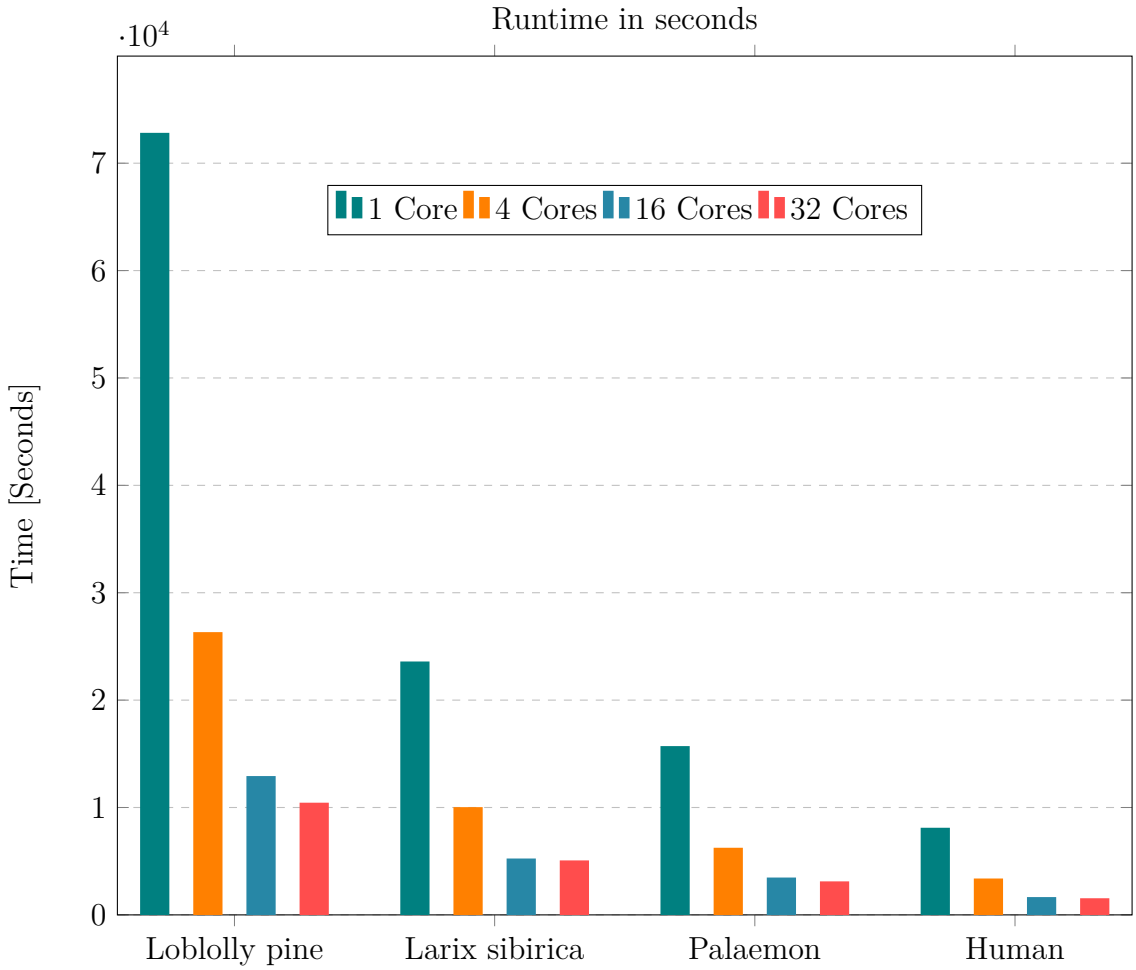


Figure 4.1: *Actual runtime in seconds for each genomes with regards to execution on 1, 4, 16 and 32 cores. Each test except Loblolly Pine 1 core has been performed multiple times and the standard deviation is accounting for the error bars present.*

input character is calculated to be roughly 24.68 bytes. While this is much higher than the WOTD-algorithm, this can reasonably be explained by the modifications made to allow for execution with larger genomes as input. This line of thought will be expanded upon in the discussion.

4.3.3 Load balancing

As described previously, because of the difference in how the data distributed, the different threads have different runtime. In figure 4.4 and 4.5 this is shown with regards to the largest genome tested. The green-blue line is the duration of the construction performed in series, the orange line in parallel and the blue line is time spent waiting. The initial observation made is that as more cores are used, the proportion of the runtime that is spent working in series grows by 25%. While a small increase is to be expected, there is more going on and this will be expanded upon in the discussion.

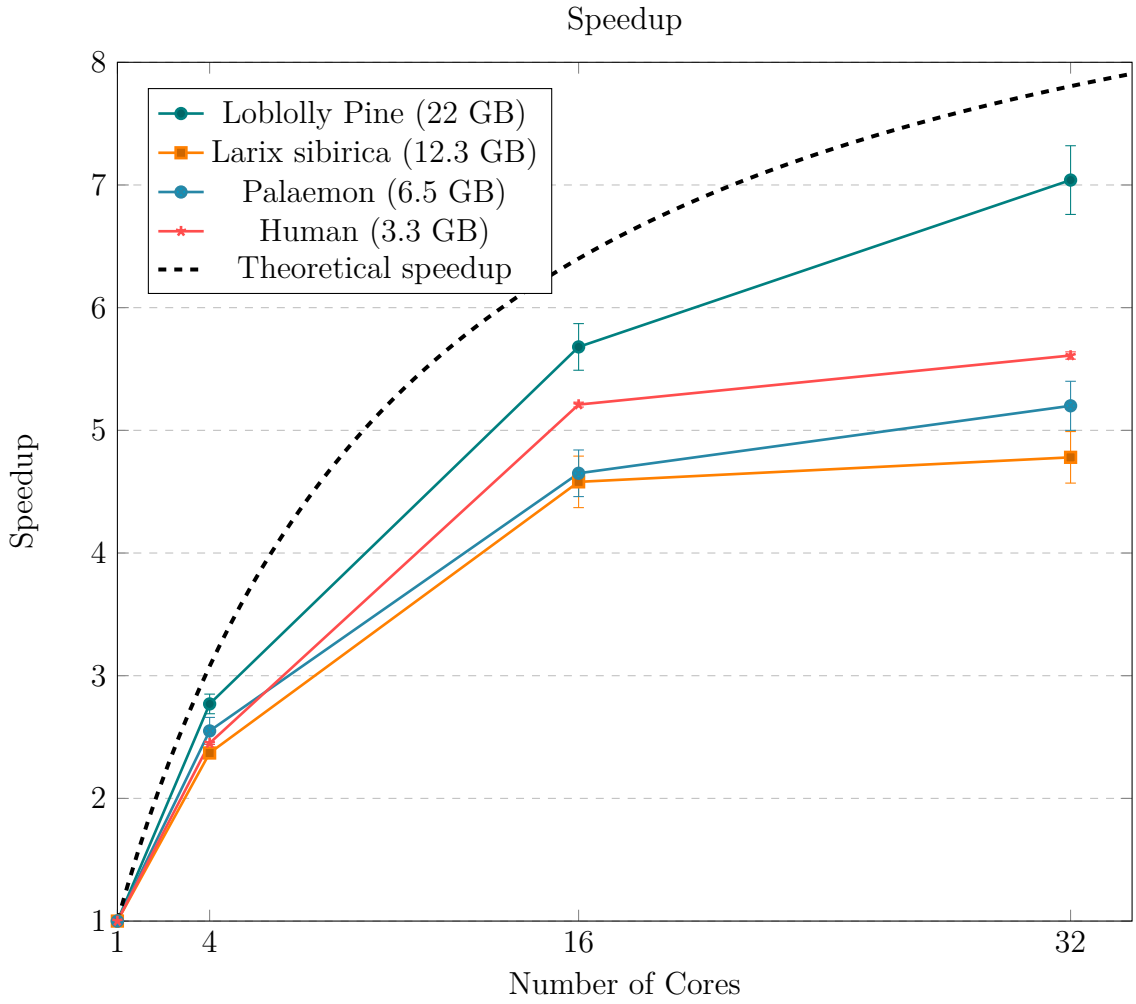


Figure 4.2: *Graph presenting the speedup, with regards to different number of cores, and how they relate to the theoretical speedup for* $p = 0.9$.

The time spent waiting increases as the number of cores increases. However, as the total execution time is shorter the actual time spent waiting is comparatively lower. The average time spent waiting when working with 4 threads is 17% and 22% for 16 cores. The distinct pattern of which threads are the fastest or slowest are very similar among the different genome. Graphs for the other 3 tested genomes can be found in appendix C.

Figure 4.3: *The peak memory consumption of the execution for the four largest genomes tested.*

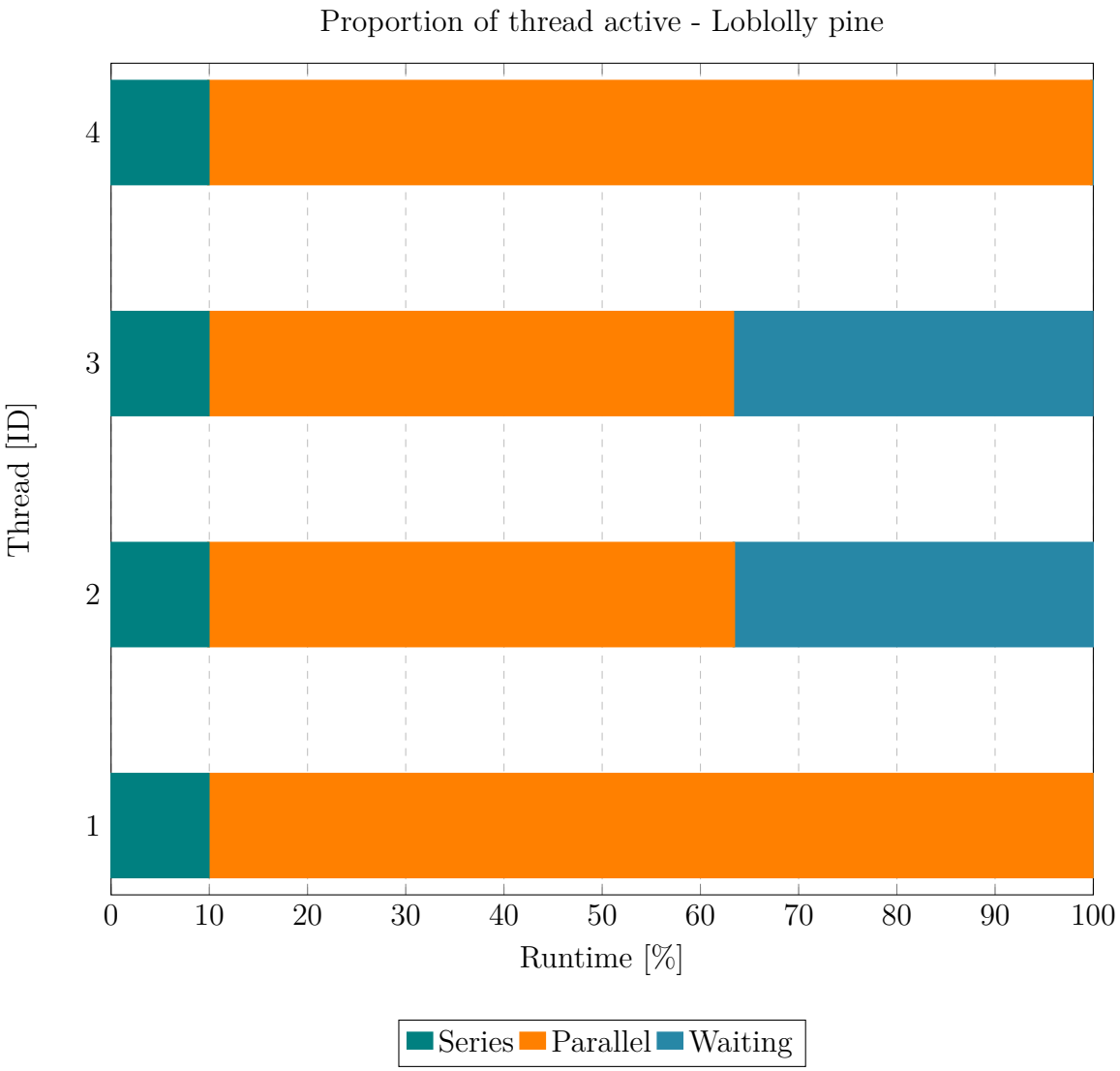


Figure 4.4: *The proportion of runtime spent in serial, parallel and the corresponding wait for the lazy suffix tree construction on 4 cores. The genome used was loblolly pine.*

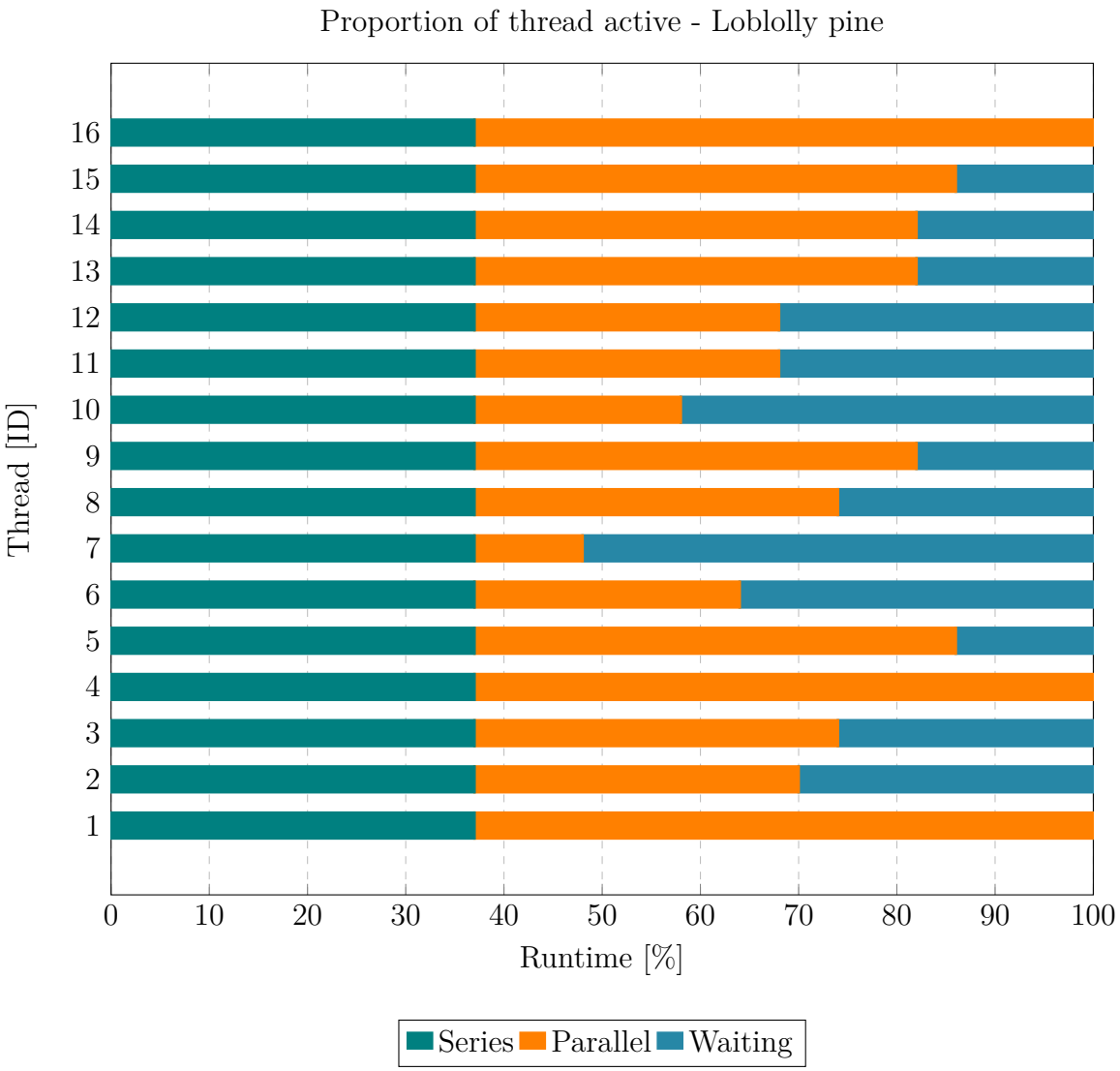


Figure 4.5: *The proportion of runtime spent in serial, parallel and the corresponding wait for the lazy suffix tree construction on 16 cores. The genome used was Loblolly pine.*

5

Discussion and Conclusion

5.1 Discussion

Following subsections discuss the results regarding the distributed and shared models.

5.1.1 Distributed Model

After exhaustive initial testing of the different parameters and their impact on the accuracy for very short genomes (<10 MB), some initial promise was shown with regards to the FSS and NLL metrics approaching 0. Based on this analysis, some relationship between the parameters and the accuracy was determined. However, intermediate tests with medium sized genomes (200-600 MB) indicated that these relationships started to break-down. Additionally, the accuracy of the parallel construction degraded with an increase of cores used and the only way to mitigate this in practice was to increase the overhead. This necessity to give the thread responsible for a sub tree construction more of the data had the effect that runtime for the higher core count was similar to that of lower core count in order to achieve the same accuracy.

Based on the observation that different genomes require different tuning to produce accuracy scores of roughly 15% (FSS 0.15, NLL 0.1-0.01) and that this approach would consume more memory lead to the investigation of other methods. As the shared memory model showed better accuracy scores, and mostly more concise results, the distributed approach was scrapped.

5.1.2 Shared Model

While the approach of distributing work on a specified depth yields some good results, the verification showed that with longer sequences some discrepancies appear. This being said, a difference of $2.32 * 10^{-}7$ for a sequence of length $387 * 10^{6}$ is to be considered very minor. More potential issues for this method are scaling, lack of load balancing and the high memory consumption, which will be discussed further in this section.

The issue with scaling in general is that as the number of cores increases the time spent in series increases proportionally. In fact, for the tests performed on 4 and 16 cores the increase is almost threefold. This problem occurs because expanding

nodes closer to the root require more work in comparison to nodes further down. Based on the trend observed, at some point the serial runtime will stand for a higher proportion than its parallel counterpart.

Because of the lack of load balancing, the effective utilisation of cores will decrease as the number of cores increases. Already, at the increment from 4 to 16 cores the portion where no threads were executing increased from 17% to 22%. This trend is expected to continue for higher numbers of available cores. There is potential to expand on this work and add load balancing properties by allowing for messages to be sent between active threads.

Lastly, the memory consumption is quite high. The main cause for this is that the large sequences considered force the program to utilise 64-bit integers rather than 32-bit for large parts of the execution. This is the only increase in comparison to the base model. While some of the additional 8 bytes could be compressed somewhat with a custom data type, this was found to be outside of the scope of the project.

Regardless of these potential issues though, the implementation showed a decent level of speedup, especially when looking at the larges genome tested. It achieves almost the theoretical speedup. The speedup for the other 3 tapers off earlier and shows little improvement between 16 and 32 cores. Something to keep in mind here is that the test shown as 32 cores is in fact 64 threads making use of 32 cores. This difference between the largest genome and the other three is most likely explained by the fact that the large input size allows the cores to be utilise better. However, it could also be because the time complexity seems to go from linear to exponential. This increased time complexity could be responsible for the extremely long runtime of the one core test, and subsequently yield higher speed-up.

Moreover, something else that could have an impact on the speedup, which has not been discussed in this report, is the architecture of the test system. An example of this in action is that in between the initial test and the tests performed to legitimise those a perceived degradation of performance on the test system led to an investigation. This investigation culminated in the changing of some system settings after which the runtime for the serial test improved. The improvement overall was approximately 15%. This is a probable cause for the large errors that can be seen in the graph shown in the result chapter.

5.2 Conclusion

This research aimed to suggest and implement parallel VLMC construction. The initial runtime analysis showed that the suffix tree construction and suffix link addition, which were the main focus of the work, accounted for 90% of the runtime. Two possible practical approaches were investigated. The so called shared model showed higher promise of performance increase and was therefore chosen as the main focus of the thesis. The results from the runtime tests showed a satisfactory level of parallelism in comparison to theoretical calculations. Additionally, it was shown that the

time-complexity approached an exponential worst case rather then the linear best case as input size grew. Lastly, the memory footprint of the implementation was determined to be very high. This and the lack of load balancing warrants future research.

Bibliography

[1] D. Dalevi, D. Dubhashi and M. Hermansson, "Bayesian classifiers for detecting HGT using fixed and variable order markov models of genomic signatures", *Bioinformatics*, vol. 22, no. 5, pp. 517-522, March 2006. [Online]. Doi: 10.1093/bioinformatics/btk029

[2] M. Borodovsky and J. McIninch, "Recognition of genes in DNA sequence with ambiguities", *Biosystems*, vol. 30, no. 1–3, pp. 161-171, 1993. [Online]. Doi: 10.1016/0303-2647(93)90068-N

[3] G. Ifrim, G. Bakir and G. Weikum, "Fast logistic regression for text categorization with variable-length n-grams", in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, Aug. 2008. [Online]. Doi: 10.1145/1401890.1401936

[4] A. Galata, N. Johnson and D. Hogg, "Learning Variable-Length Markov Models of Behavior", *Computer Vision and Image Understanding*, vol. 81, no. 3, pp. 398-413, March 2001. [Online]. Doi: 10.1006/cviu.2000.0894.

[5] D. S. Fava, S. R. Byers and S. J. Yang, "Projecting Cyberattacks Through Variable-Length Markov Models", in *IEEE Transactions on Information Forensics and Security*, vol. 3, no. 3, pp. 359-369, Sept. 2008.

[6] R. Giegerich, S. Kurtz and J. Stoye, "Efficient implementation of lazy suffix trees", *Software: Practice and Experience*, vol. 33, no. 11, pp. 1035-1049, March 2003.

[7] M. G. Maaß, "Computing suffix links for suffix trees and arrays", *Information Processing Letters*, vol. 101, no. 6, pp. 250-254, March 2007. [Online]. Doi: 10.1016/j.ipl.2005.12.012.

[8] M. H. Schulz, D. Weese, T. Rausch, A. Döring, K. Reinert, and M. Vingron, "Fast and adaptive variable order Markov chain construction", in *International Workshop on Algorithms in Bioinformatics* pp. 306-317. Sept. 2008

[9] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*. Place of Publication: Cambridge University Press, 1997.

[10] E. Ukkonen, "On-line construction of suffix trees", *Algorithmica*, vol. 14, no. 3, pp. 249-260, Sept. 1995. [Online]. Doi: 10.1007/BF01206331

[11] E. M. McCreight, "A space-economical suffix tree construction algorithm", *Journal of the ACM*, vol. 23, no. 2, pp. 262-272, April 1976. [Online]. Doi: 10.1145/321941.321946

[12] J. Gustafsson, E. Norlander, "Clustering genomic signatures A new distance measure for variable length Markov chains", Chalmers University of Technology / Department of Computer Science and Engineering (Chalmers), 2018. [Online]. Available: https://hdl.handle.net/20.500.12380/255511

[13] Multiprocessing-Process-based Parallelism—Python 3.8.2 Documentation. Retrieved March. 2020. Available: https://docs.python.org/3/library/multiprocessing.html
[14] . C++11 std::thread Libaray. Retrieved March. 2020. Available: https://devdocs.io/cPages/thread/thread
[15] C++ Chrono Libaray. Retrieved March. 2020. Available: https://devdocs.io/cPages/header/chrono

A

Hardware information of test system

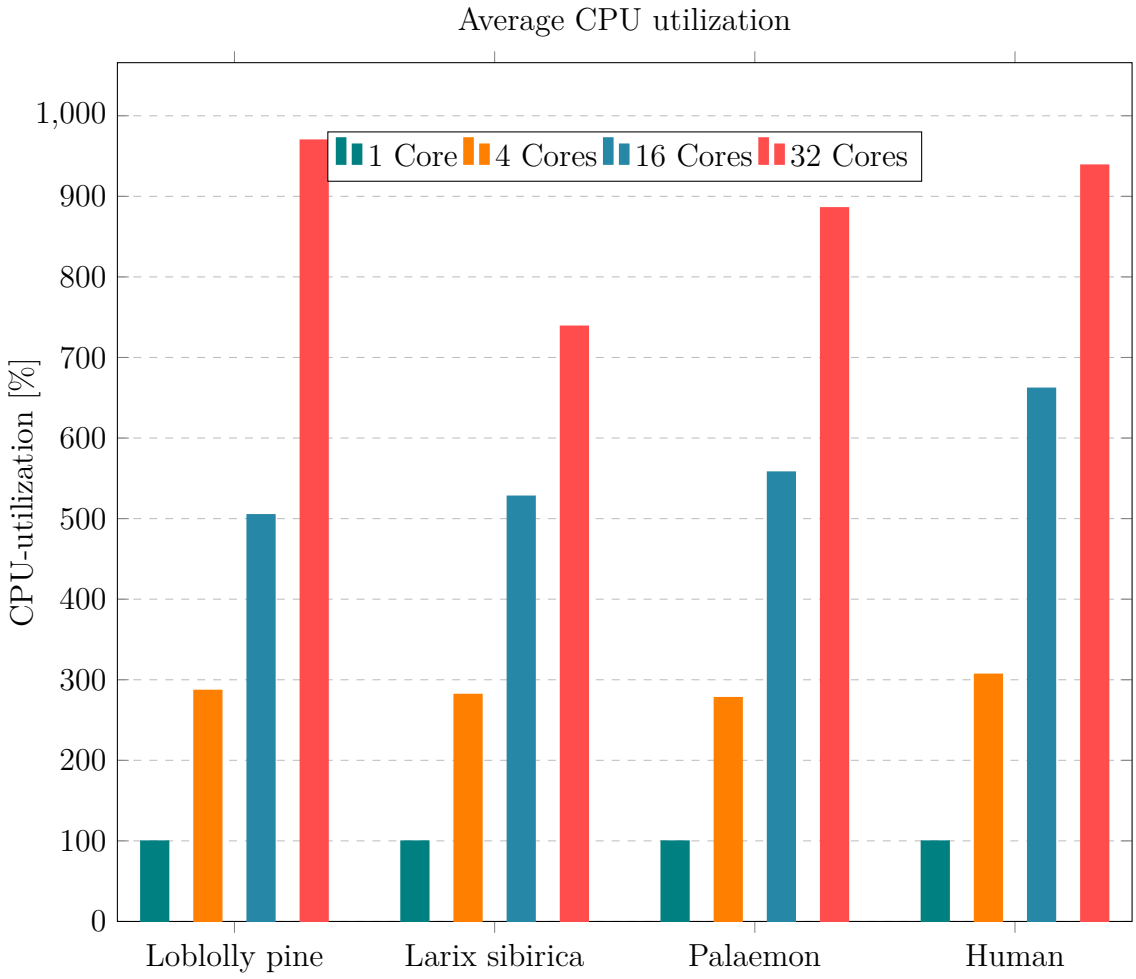
```
cat /proc/cpuinfo

processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
stepping        : 4
microcode       : 0x2000064
cpu MHz         : 3630.602
cache size      : 19712 KB
physical id     : 0
siblings        : 24
core id         : 0
cpu cores       : 12
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
                  pge mca cmov pat pse36 clflush dts acpi mmx fxsr s$
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass
                  l1tf mds
bogomips        : 5200.00
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
...

cat /proc/meminfo

MemTotal:       791275468 kB
```

```
MemFree:         687107596 kB
MemAvailable:    757022628 kB
Buffers:           986128 kB
Cached:          68532740 kB
SwapCached:            0 kB
Active:          78343352 kB
Inactive:        13755588 kB
Active(anon):    22582276 kB
Inactive(anon):    117152 kB
Active(file):    55761076 kB
Inactive(file):  13638436 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:        515068 kB
SwapFree:         515068 kB
Dirty:                 4 kB
Writeback:             0 kB
AnonPages:       22573568 kB
Mapped:            705992 kB
Shmem:             119364 kB
Slab:             6636028 kB
SReclaimable:     5502528 kB
SUnreclaim:       1133500 kB
KernelStack:        24464 kB
PageTables:        433804 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     396152800 kB
Committed_AS:    133708596 kB
VmallocTotal:    34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:    7366656 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
CmaTotal:              0 kB
CmaFree:               0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:     3655488 kB
DirectMap2M:     529747968 kB
```

DirectMap1G: 272629760 kB

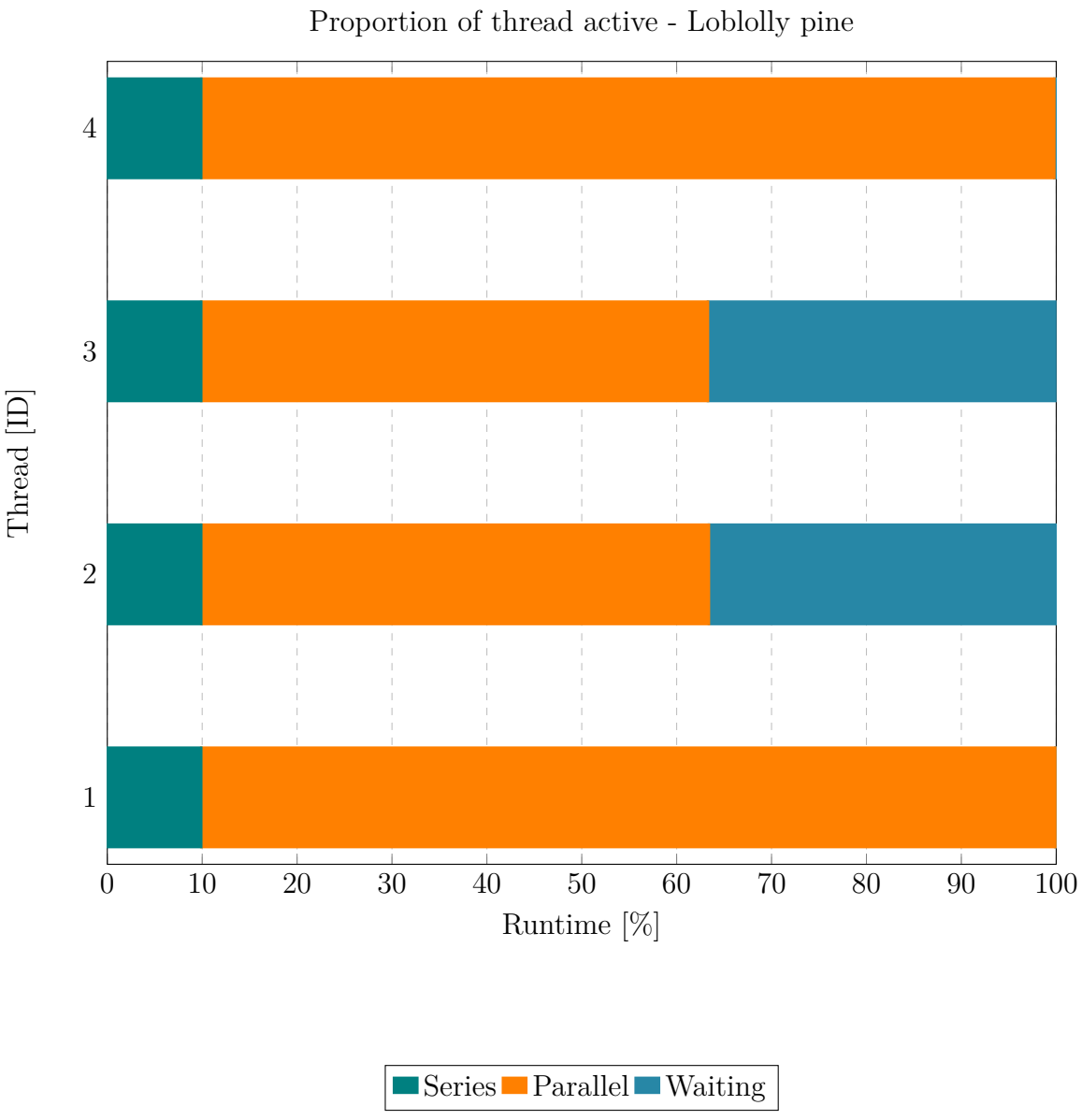
B

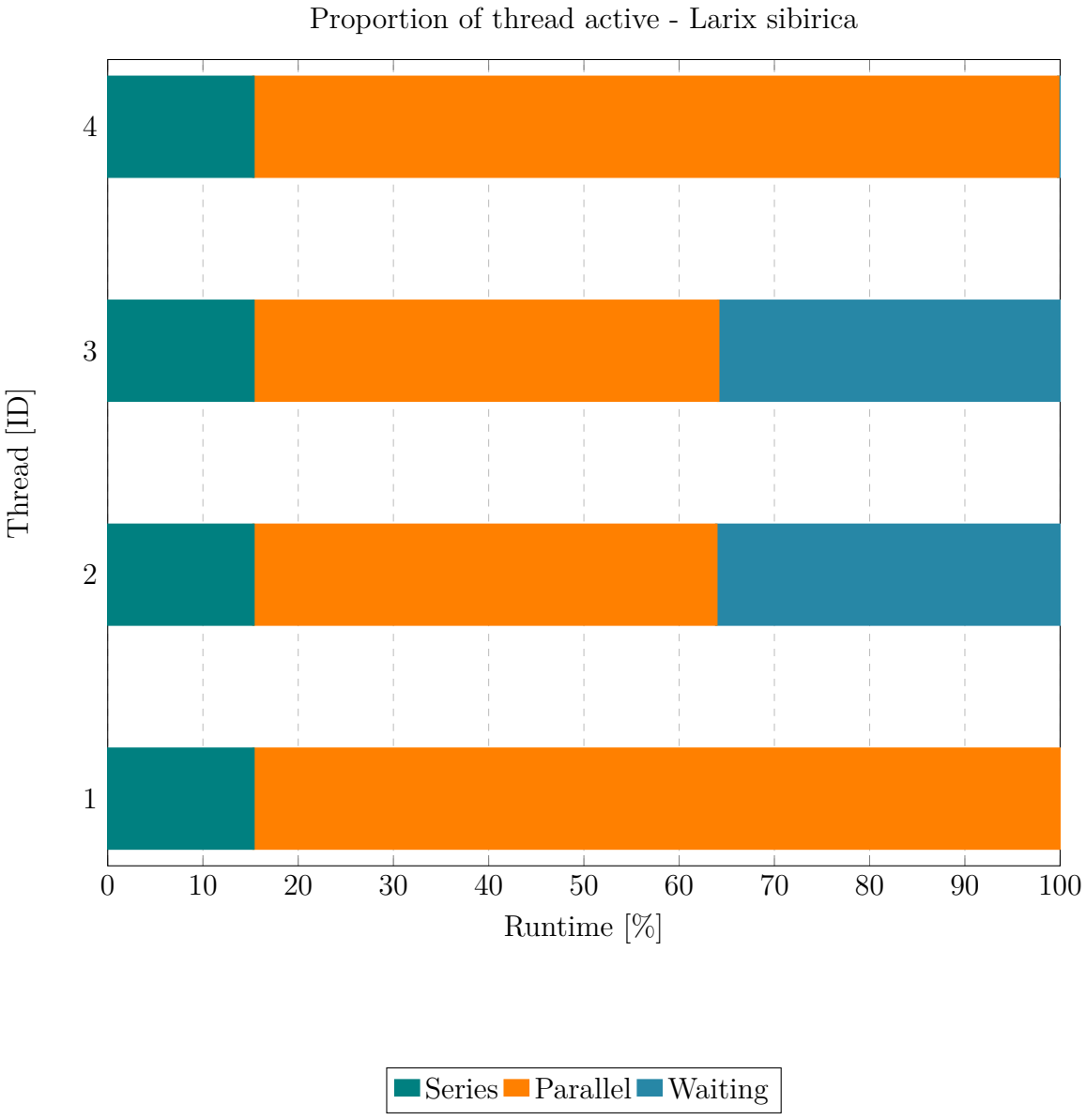
CPU utilisation

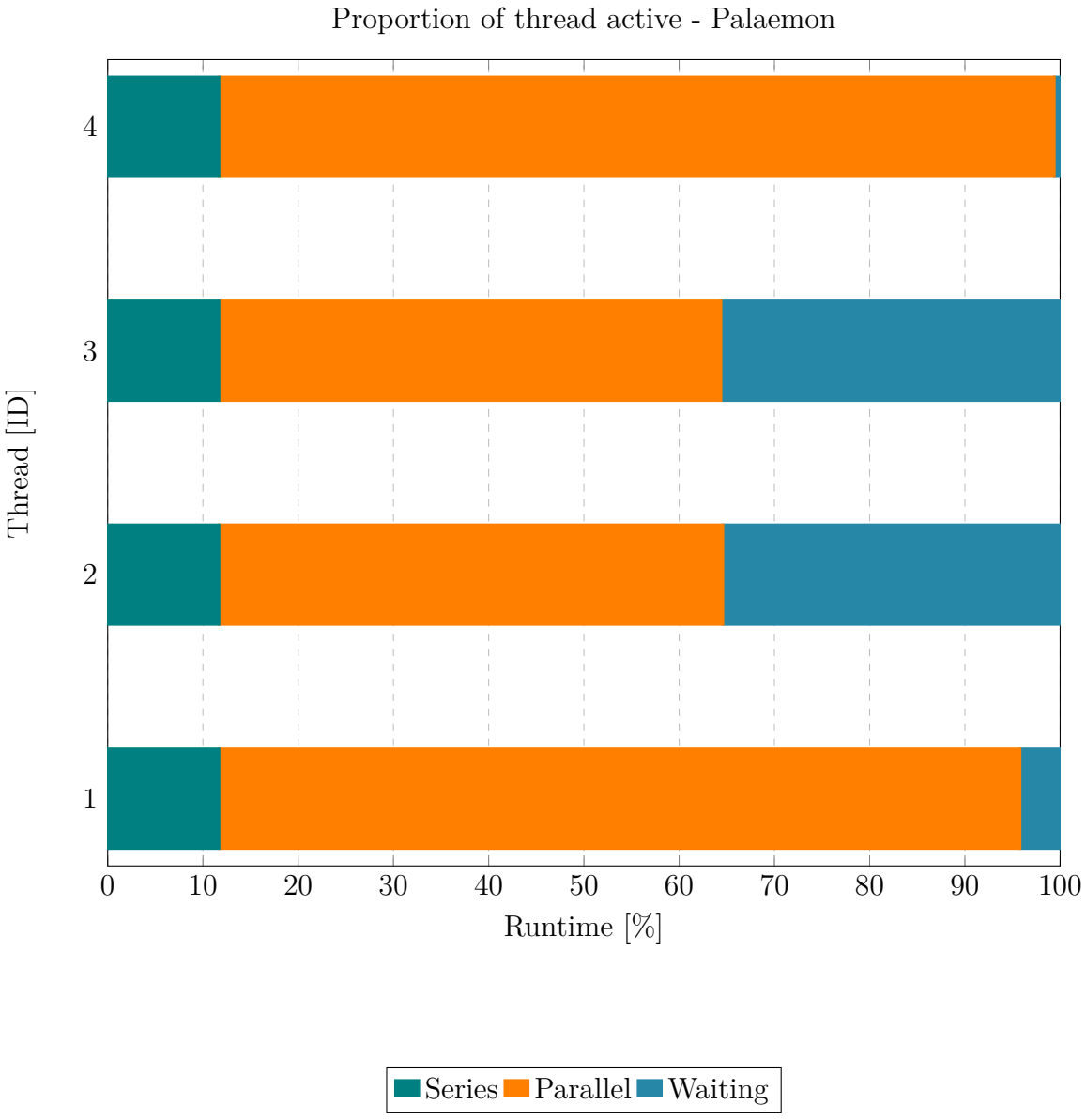


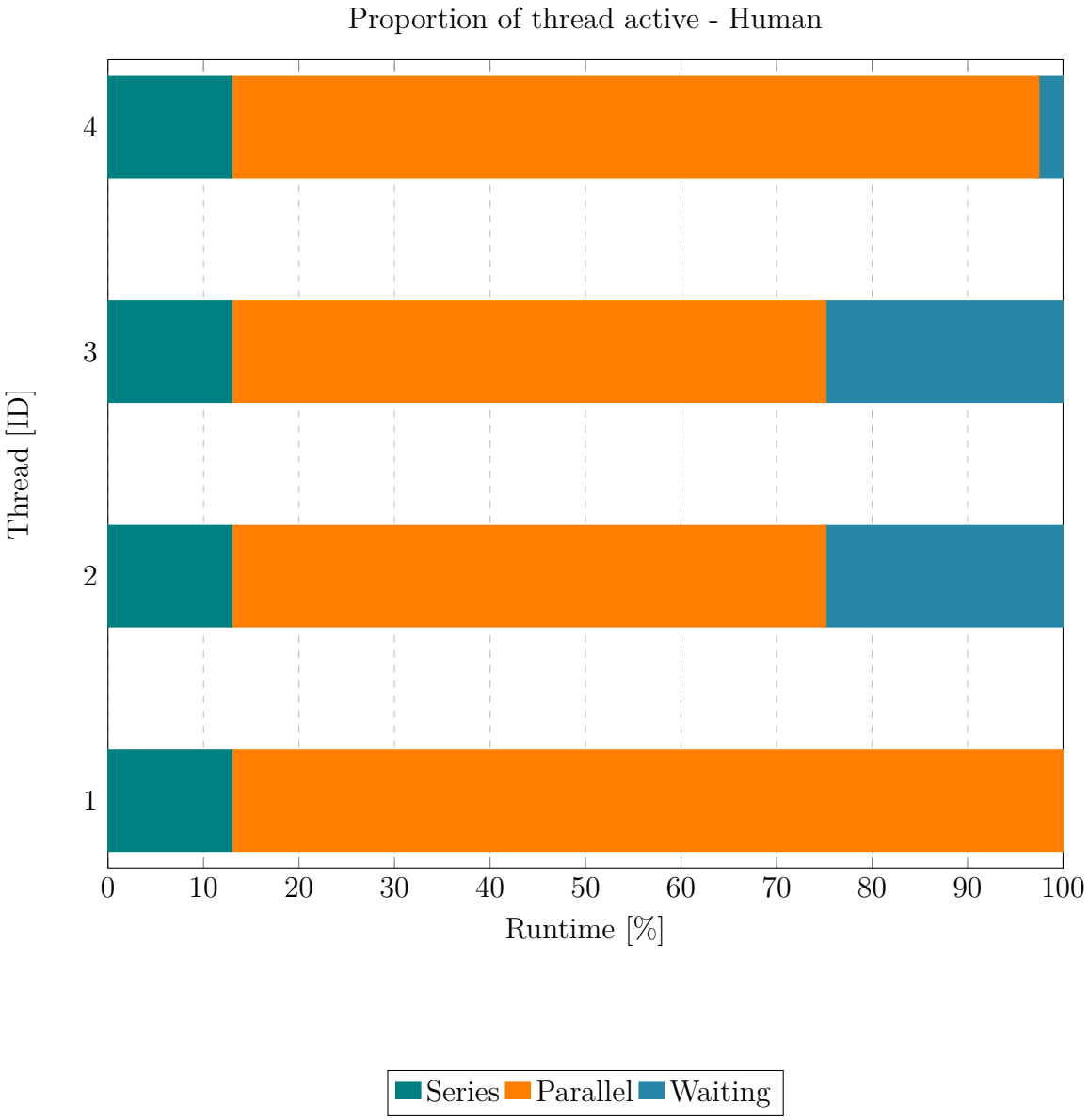
C

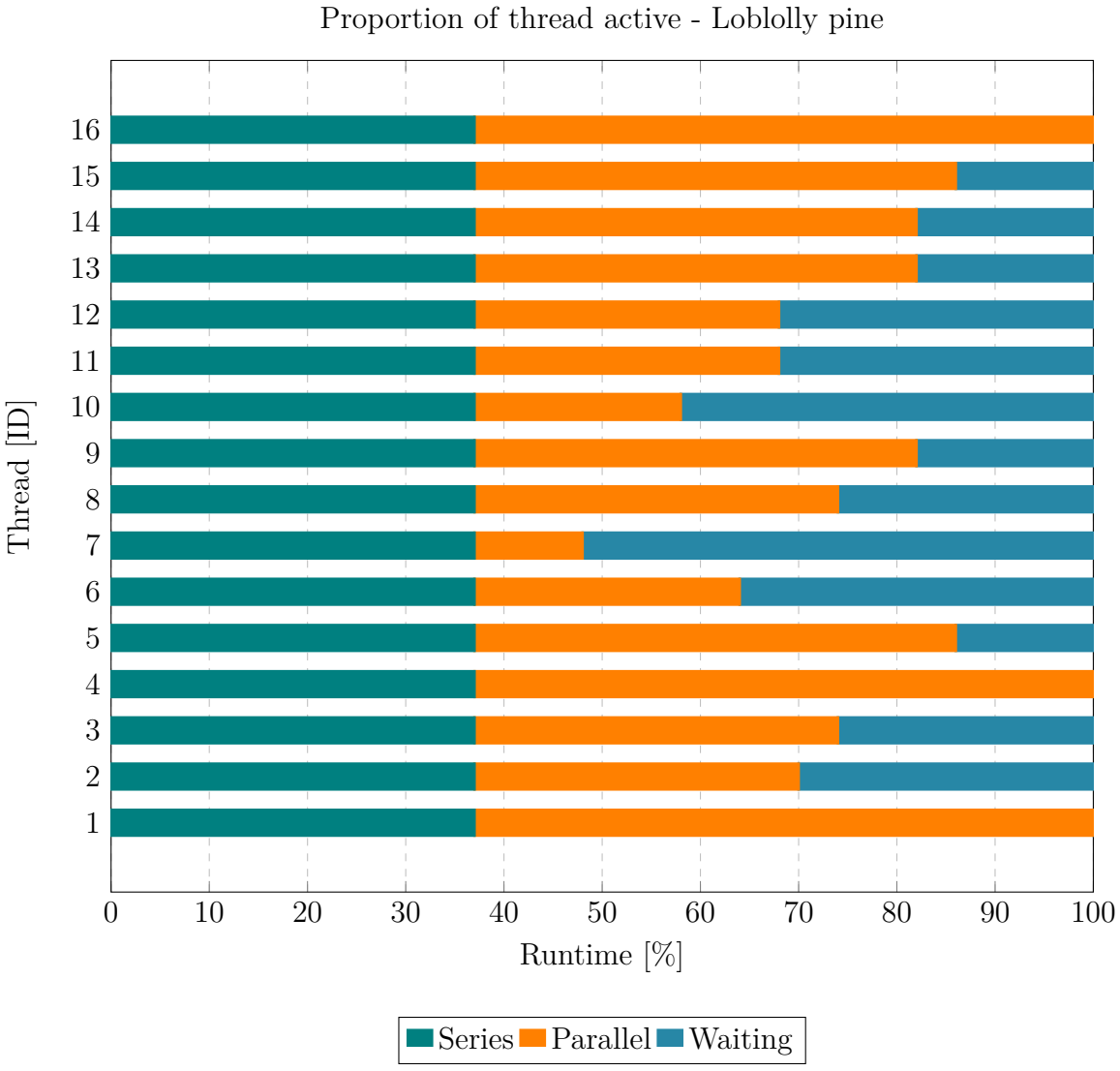
Visualisation of thread execution

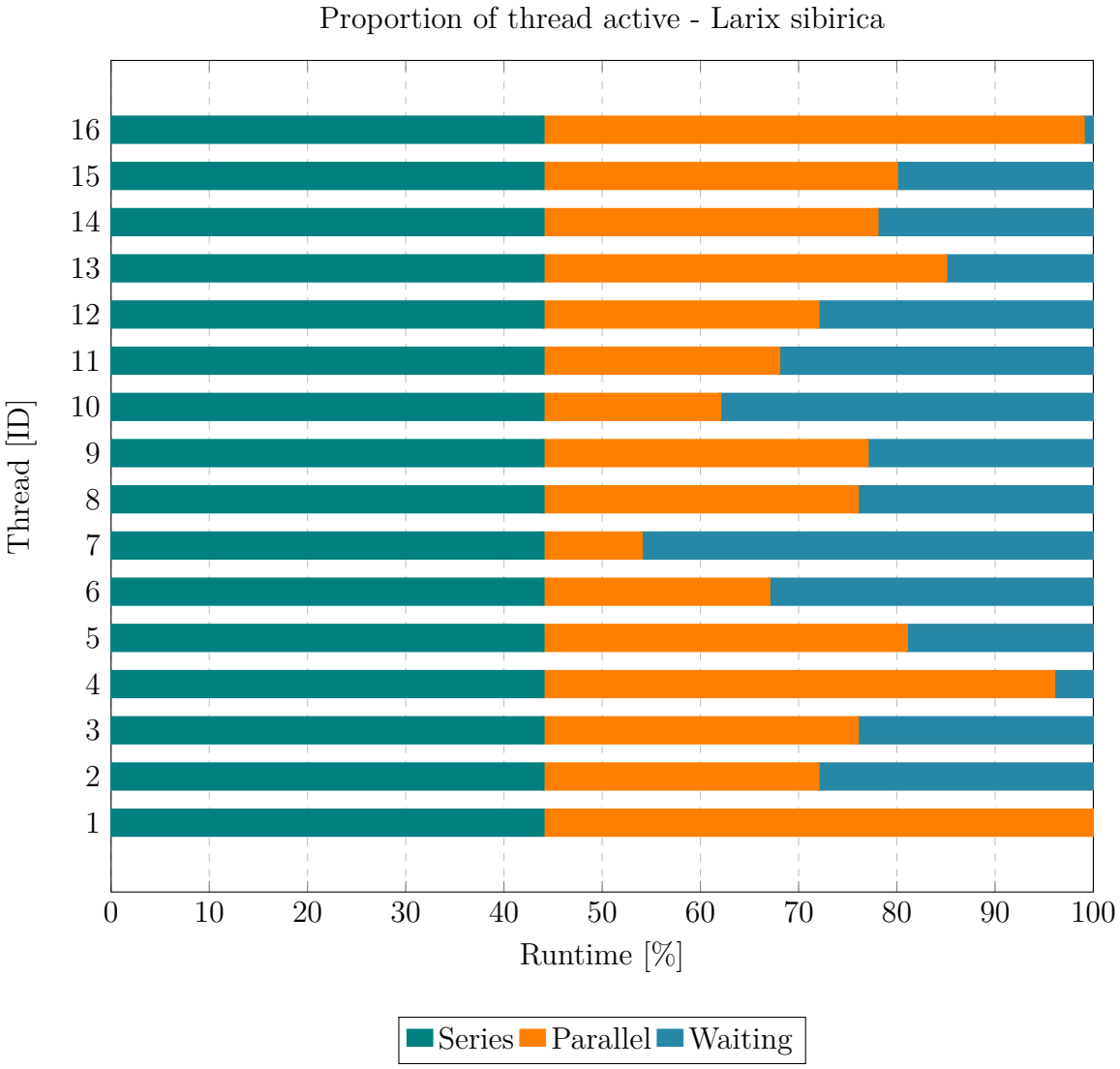


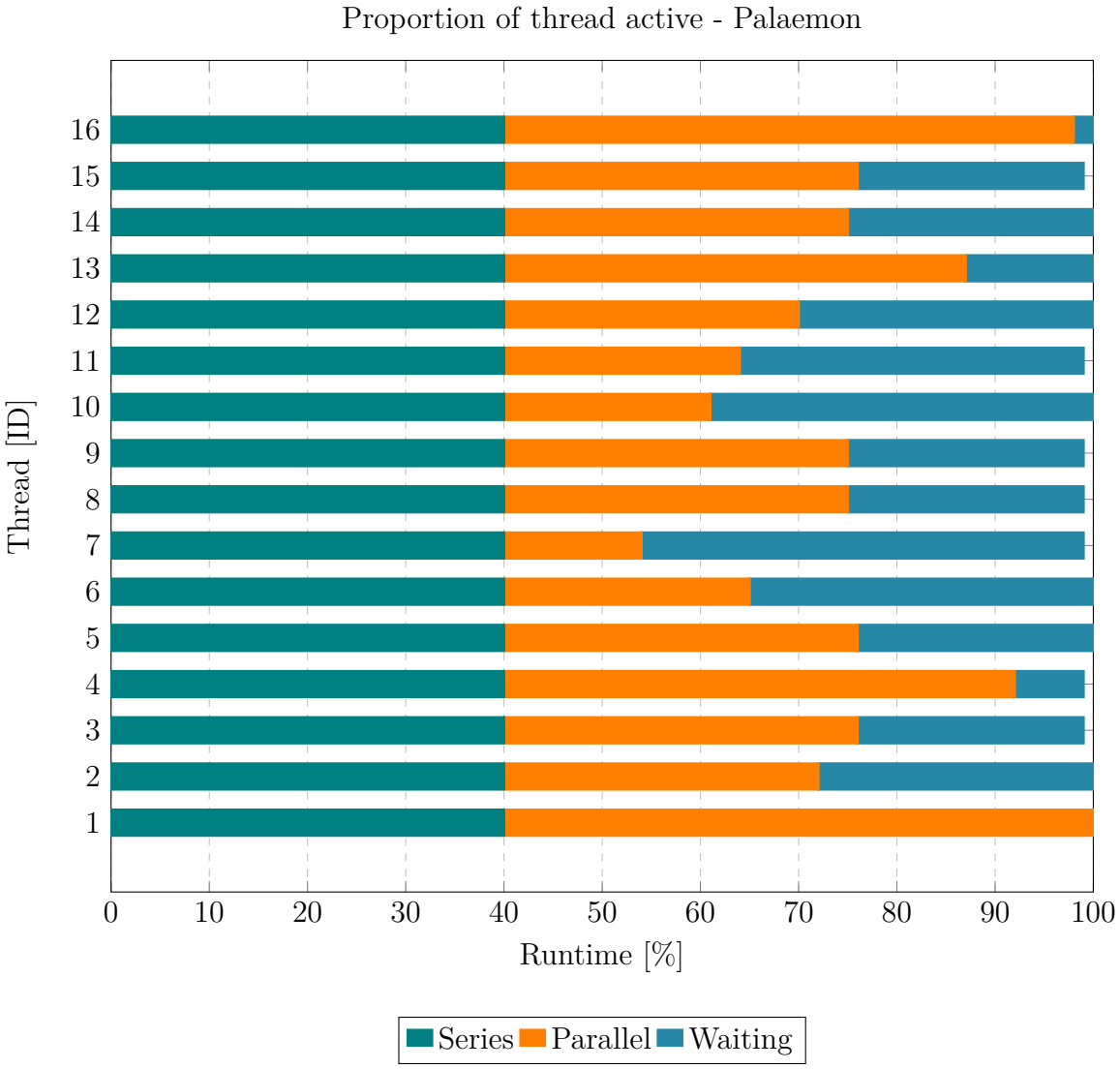

Proportion of thread active - Larix sibirica
Thread [ID]
4
3
2
1
0
10
20
30
40
50
60
70
80
90
100
Runtime [%]
Series
Parallel
Waiting

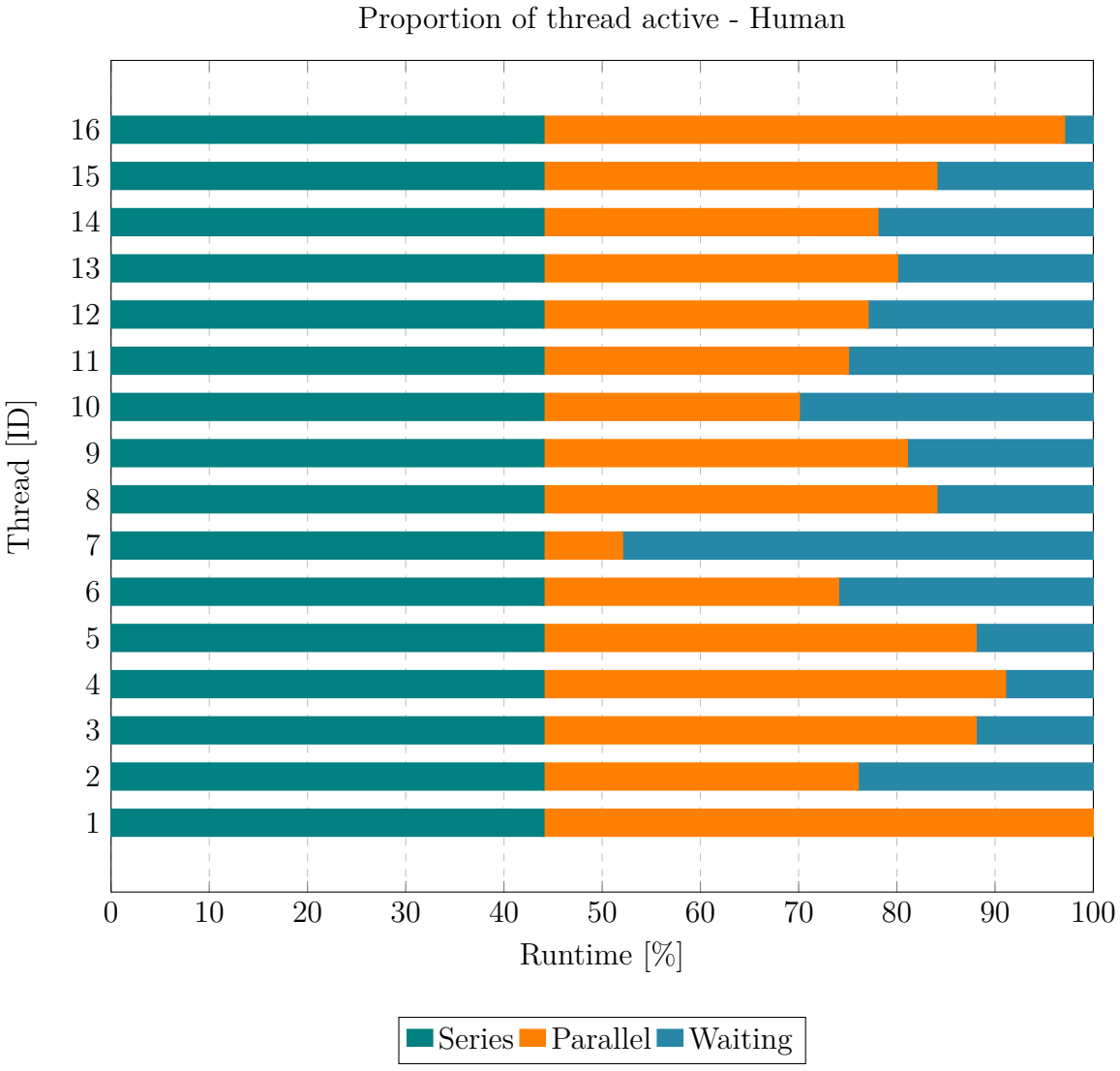

Proportion of thread active - Palaemon
Thread [ID]
4
3
2
1
0
10
20
30
40
50
60
70
80
90
100
Runtime [%]
Series
Parallel
Waiting


Proportion of thread active - Human
Thread [ID]
4
3
2
1
0
10
20
30
40
50
60
70
80
90
100
Runtime [%]
Series
Parallel
Waiting


Proportion of thread active - Loblolly pine
Thread [ID]
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
10
20
30
40
50
60
70
80
90
100
Runtime [%]
Series
Parallel
Waiting


Proportion of thread active - Larix sibirica
Thread [ID]
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
10
20
30
40
50
60
70
80
90
100
Runtime [%]
Series
Parallel
Waiting


Proportion of thread active - Palaemon
Thread [ID]
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
10
20
30
40
50
60
70
80
90
100
Runtime [%]
Series
Parallel
Waiting


Proportion of thread active - Human
Thread [ID]
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
10
20
30
40
50
60
70
80
90
100
Runtime [%]
Series
Parallel
Waiting