Optimization of a Feature Selection Tool for Inference of Gene Regulatory Networks

João Vítor Fuzetti da Cunha

Dissertation presented to the School of Technology and Management of Bragança to obtain the master's degree in Informatics within the scope of the double degree program with the Federal University of Technology – Paraná.

Supervisors:
Fabrício Martins Lopes (UTFPR)
José Carlos Rufino Amaro (IPB)

Bragança
November 2025

Optimization of a Feature Selection Tool for Inference of Gene Regulatory Networks

João Vítor Fuzetti da Cunha

Dissertation presented to the School of Technology and Management of Bragança to obtain the master's degree in Informatics within the scope of the double degree program with the Federal University of Technology – Paraná.

Supervisors:

Fabrício Martins Lopes (UTFPR)

José Carlos Rufino Amaro (IPB)

Bragança

November 2025

Acknowledgment

To my advisors, for their attentive guidance and dedication throughout the development of this dissertation. I thank them for their patience with my questions, for their support in the face of challenges that arose along the way, and for the valuable contributions that helped me maintain the organization and direction of the work.

To my family, for their unconditional support and constant encouragement to continue my studies. Despite the distance, I always felt the affection and presence of each one of you, who cheered me on throughout this journey.

To my girlfriend and my friends, for their companionship, understanding, and constant encouragement to keep me progressing, even in the face of daily difficulties. To all of you, I thank you for the lightness you brought to the most intense moments and for the shared strength throughout this process. To my colleagues, already included, for their empathy and support during the periods when I needed to balance professional demands with my studies, without ever ceasing to motivate me. To my everyday friends, for not letting me miss out on every free moment, whether in person or remotely, chatting idly. To my girlfriend, especially, for her constant patience, affection, and encouragement, which were fundamental in keeping me strong until the end of this journey.

To the UTFR and IPB institutions, for the opportunity to develop this work and for the learning and growth environment provided. The institutional support and conditions offered were essential for the completion of this dissertation.

Abstract

This dissertation concerns the computational optimization of DimReduction, a feature selection tool for inference of Gene Regulatory Networks (GRN). The primary aim was to make it faster and more performance scalable, in order to to handle large amounts of data, which would bring it closer to the bioinformatics community. The approach involved the translation of the original Java GUI-based implementation into a CLI version and the re-implementation of the latter in Python. Once the performance of the Python version was lower than expected, the focus turned again to the Java CLI version. The major bottleneck in this version was determined and addressed, namely the elimination of explicit invocation of the Garbage Collector (GC) led to the runtime of a reference dataset (with 4511 genes) to reduce from more than 2 days to 42 minutes. The optimized version of Java was then parallelized, using a threaded approach, which yielded near-linear speedups. The new Java parallel implementation was then compared with other reference platforms from the literature (GENIE3, CLR, ARACNE, C3NET, BC3NET, MRNET, MRNETB, KBOOST and PCIT). The findings indicate that even though some alternatives ensure higher metrics of quality (AUROC/AUPR), DimReduction speed makes it a competitive tool in the field.

Resumo

Esta dissertação aborda a otimização computacional do DimReduction, uma ferramenta de seleção de características para inferência de Redes de Regulação Gênica (GRN). O objetivo principal foi torná-la mais rápida e com melhor escalabilidade de desempenho, para lidar com grandes quantidades de dados, aproximando-a da comunidade bioinformática. A abordagem envolveu a tradução da implementação original em Java com interface gráfica para uma versão de linha de comando (CLI) e a reimplementação desta última em Python. Como o desempenho da versão em Python ficou abaixo do esperado, o foco voltou-se para a versão em Java CLI. O principal gargalo desta versão foi identificado e resolvido: a eliminação da invocação explícita do Garbage Collector (GC) levou à redução do tempo de execução de um conjunto de dados de referência (com 4511 genes) de mais de 2 dias para 42 minutos. A versão otimizada em Java foi então paralelizada, utilizando uma abordagem com threads, o que resultou em ganhos de velocidade quase lineares. A nova implementação paralela em Java foi então comparada com outras plataformas de referência da literatura (GENIE3, CLR, ARACNE, C3NET, BC3NET, MRNET, MRNETB, KBOOST e PCIT). Os resultados indicam que, embora algumas alternativas apresentem métricas de qualidade superiores (AUROC/AUPR), a velocidade do DimReduction o torna uma ferramenta competitiva na área.

Contents

List of Figures

Acronyms

API Application Programming Interface.

CLI Command Line Interface.

DB Database.

DBMS Database Management System.

ESTiG Escola Superior de Tecnologia e Gestão.

GC Garbage Collector.

GIL Global Interpreter Lock.

GUI Graphical User Interface.

HTTP HyperText Transfer Protocol.

IPB Instituto Politécnico de Bragança.

RDBMS Relational Database Management System.

Chapter 1

Introduction

Genomic sequence analysis has become one of the pillars of contemporary biology, with considerable implications to our well-being and life in general. As the high-throughput sequencing technologies keep on advancing incessantly, Bioinformatics is continuously producing a flood of genomic data and this situation poses serious computational necessity.

Unraveling the relationship between genes, which is a computation procedure called Gene Regulatory Network (GRN) inference, is important in learning complex biological processes, including cell differentiation and disease psychopathology. Yet, the dimension of the genomic data used (high dimensionality) makes this task extremely difficult with the number of features (genes) possibly reaching thousands, whereas the number of samples is minimal, a basic issue in bioinformatics known as the "curse of dimensionality".

In order to explore this large feature space, one important approach to narrow it down is applying Feature Selection (FS) to select only the most relevant features (genes). As a result, the issue of inferring GRNs given expression data is represented as a series of FS problems. The open-source software DimReduction [1] provides an environment to accomplish this task, and it uses algorithms of criterion search. However, in its original form, as a Java application with a GUI, DimReduction has some practical drawbacks that prevent it to be used as a large-scale modern scientific applications, such as not being fully automated and not being prepared to exploit parallel architectures to accelerate its execution, thus requiring considerable time in handling large and real world-scale datasets.

For instance, using real data from the DREAM5 Network Inference Challenge, namely Network 3 (a reference dataset of Escherichia coli with 4511 genes), consumes more than two days of processing time.

Minimizing the processing time of GRN inferences, however, may come at the cost of the quality of such inference. The ideal inference method would be the one that simultaneously minimizes the processing time and maximizes the quality of the inferences or predictions. However, in a real-work scenario, a tradeoff between these two factors must always be considered. For that reason, it is also important to study the specific tradeoffs of different inference methods (including the one that is the focus of this study).

1.1 Motivation, Objectives and Contributions

Having a personal interest in the fields of Bioinformatics and High Performance Computing, the possibility of reworking the DimReduction tool, so that it could be more widely recognized and used, and at the same time being more performant, was highly motivating.

To foster DimReduction's adoption, the initial plan was to translate the original Java code to Python, due to the current prevalence of this language in the software ecosystem that nowadays serves the Bioinformatics community. It was also expected that the re-implementation in Python could be an opportunity to improve the performance of DimReduction, namely through parallelization, making it ready to exploit modern multicore architectures and, at the same time, having a performance leap over the old Java version.

However, after extensive analysis and profiling of the newly developed Python CLI-based version, together with a new CLI-based variant of the original Java code, the performance of the new Python code was found consistently behind Java's, even when employing threaded parallelism. For that reason, the focus turned back to the improvement of the new CLI-based Java version, also employing multithreading. A final comparison with alternative methods from the literature, in terms of performance and inference quality, highlighted the merits of the new parallel Java-based DimReduction implementation.

1.2 Thesis Structure

This dissertation has four major chapters which are arranged after this Introduction.

Chapter 2 provides the theoretical background, which essentially highlights the main concepts of Feature Selection algorithms, the scenario of the DREAM challenges, the associated GRN inference algorithms, and the core concepts of parallelism for this thesis.

Chapter 3 outlines the performance analysis methodology, applied to newly developed CLI-based versions of DimReduction in Java and Python, discovering performance bottlenecks using profiling, and improving Python's performance through parallelization.

Chapter 4 leverages the new CLI-based Java code through parallelization, and shows benchmarking results that showcase high performance and strong scalability; it also provides a comparison with reference methods from the literature, in terms of predictive quality, execution time and a weighted average of both.

Chapter 5 includes the generalizations made by the work, where the optimization methodology and the competitiveness of the tool is the foremost result; some directions towards future research, such as validation of other internal algorithms and integration with the GENECI project, are proposed.

Chapter 2

Related Concepts and Methods

This chapter elaborates the theoretical concepts and methods that are very important in the optimization of the DimReduction tool. It addresses entropy within the context of information theory, advanced feature selection criteria and algorithms, the Gene Regulatory Network inference problems (DREAM), a survey of similar methods, and sums up the main concepts related to parallelism in the context of high-performance computing.

2.1 Entropy in Information Theory

The Information Theory, which was pioneered by Shannon, is frequently used in the selection of features in genomic data [1] [2]. Entropy, in this case, is a basic way of measuring the uncertainty of a variable [2]. The greater the value of entropy, the more the uncertainty when attempting to predict the value of a given variable [2].

In terms of the probabilities $P(x)$ of its possible occurrences, the entropy of a discrete random variable X is mathematically defined as Shannon's entropy $H(X)$ [2],

$$H(X) = -\sum_{x \in X} P(x) \log P(x), \tag{2.1}$$

in which the summation is over all possible values x which X can take. This is an indicator of the disorder or uncertainty in a dataset, which is used to grade it [2].

2.1.1 Generalized Tsallis Entropy

In order to effectively model complex systems, such as Gene Regulatory Networks (GRNs), in which uncertainty and long-range correlations may be present, the standard measure of uncertainty, obtained in the works of Shannon, called Boltzmann-Gibbs statistics, can be generalized [3]. This generalization is called the **Tsallis Generalized Entropy**, and it includes one more parameter, namely, q, which is used to describe the degree of non-extensivity of the system [1] [2]. Its mathematical formulation is given by:

$$H_q(X) = \frac{1 - \sum_{x \in X} P(x)^q}{q - 1}, \tag{2.2}$$

where X is a discrete random variable, $P(x)$ is the probability of outcome x for X, and q is the **entropic parameter** ($q \in \mathbb{R}$) – the measure of non-extensivity of the system.

An important property of the Tsallis entropy is that if the entropic parameter, q, tends to 1, the entropy of the system ($H(X)$) is recovered, i.e., $H_1(X) = H(X)$. The entropic properties are determined by the value of q: $q = 1$ defines an extensive system (additivity), $q < 1$ defines a super-extensive system (super-additivity) and $q > 1$ defines a sub-extensive system (sub-additivity) [1] [2].

Since the Tsallis entropy can be adapted to feature selection, with variants, such as Mean Conditional Entropy (MCE), it can be used to explore the question of whether GRNs are non-extensive in nature. Experimental results in network inference indicate that with sub-extensive entropy (such as with $q > 1$), for example in the range of $q \approx 2.5$, one can substantially predict better and minimize false positives, than with more classical Shannon entropy (with $q = 1$) [2].

2.2 Mean Conditional Entropy

In feature selection, in problems of classification or network inference, the objective is frequently to measure the dependence of a collection of features (X) on a target variable

(Y). To this end, the concept of the so-called **Conditional Entropy** is essential. Conditional entropy, denoted as $H(Y|x)$, is a measure of the uncertainty left in the variable Y when an example x of X has been observed [2]. The **Mean Conditional Entropy** (MCE), $H(Y|X)$, generalizes this concept, by taking the weighted average of the conditional entropies over all possible instances $x \in X$. The MCE is given by 2.3 [1], [2]:

$$H(Y|X) = \sum_{x \in X} P(x)H(Y|x), \tag{2.3}$$

where, Y is the target variable (gene), X is the set of observed instances (predictors/features), $P(x)$ is the probability of observing instance x, and $H(Y|x)$ is the conditional entropy of a variable Y given the observation of an X, which is calculated as $H(Y|x) = -\sum_{y \in Y} P(y|x) \log P(y|x)$ [2].

The goal of applying MCE as a feature selection criterion is minimization: **the smaller the value of the $H(Y|X)$, the better a feature subspace is**, since the value represents the greater amount of information about Y that the observation of X provides [1] [2].

2.2.1 Sequential Floating Forward Selection Algorithm

The computational search stage plays a vital role in identifying the best subset of features that optimize the selected criterion function (such as MCE). An exhaustive search is optimal, but too prohibitive mathematically to compute for high-dimensional bioinformatics problems. Therefore, sub-optimal heuristic algorithms are normally used [1] [2].

The **Sequential Floating Forward Selection (SFFS)** algorithm is one of the strong sub-optimal methods. It was developed to address the undesirable "nesting effect" that is present in more simplistic algorithms such as Sequential Forward Selection (SFS) [1] [2].

SFFS Operation

The SFFS algorithm is a combination of two simpler searching methodologies, SFS and Sequential Backward Selection (SBS), repeated a number of times [1]. As shown in Figure 2.1, it can be described to be floating in the following steps [1] [2]:

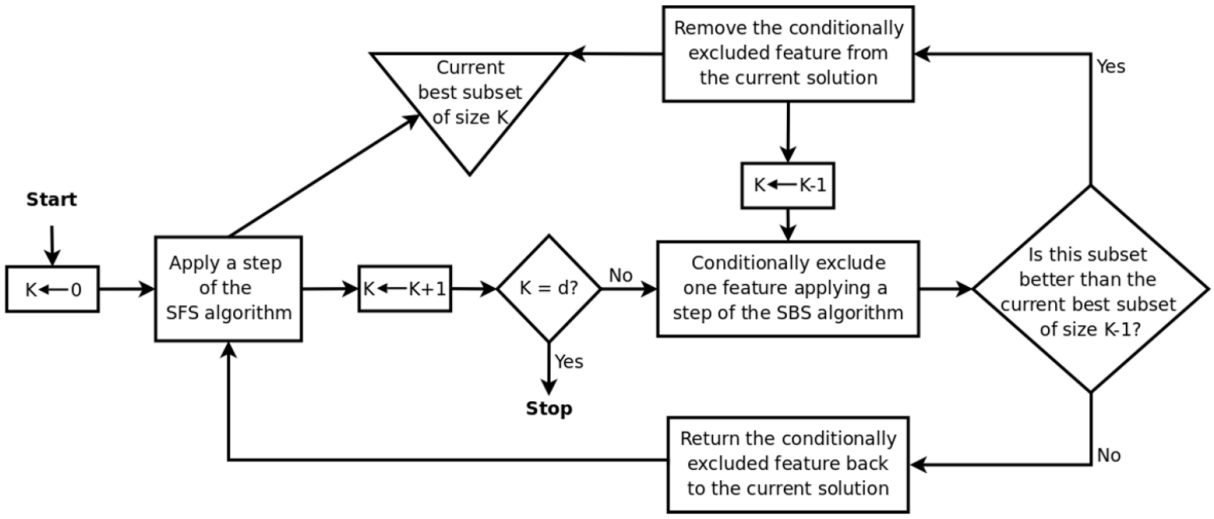


Figure 2.1: Simplified flowchart of the SFFS algorithm from Lopes, Martins, and Cesar [1] and Lopes and César Júnior [2] (adapted from Somol, Pudil, Novovičová, *et al.* [4]). K is the size of the subset of the current solution and d is the size of the subset of the final solution being desired (algorithm stopping condition).

1. **Initial Forward Steps (SFS):** The algorithm starts with a feature set of cardinality 0 (where the subset cardinality, K, is 0). It then uses Sequential Forward Selection (SFS) which implies adding the single best remaining feature at each iteration step, and the feature should optimally enhance the selected criterion function. This forward search proceeds further until the subset size is at $K = 2$.

2. **Conditional Exclusion (SBS):** When the subset size becomes more than two ($K > 2$), the algorithm begins a conditional exclusion step, which is essentially the Sequential Backward Selection (SBS) method. SBS repeatedly removes features. Here the aim is to test whether any of the existing features is removable with an improved subset as per the criterion function.

3. **Floating Search and Storage:** The SFFS switches between adding the best remaining feature (SFS step) and removing the worst feature in case the removal would benefit the set (SBS step). This permits features which have been previously chosen to be eliminated in the case they become unnecessary or harmful in combination with new features, thus avoiding the nesting effect .

4. **Stopping Condition:** The process is continued until a stopping condition is achieved (usually determined by a desired maximum size of a subset or a performance limit).

5. **Result:** The intermediate results (the best sets found at each cardinality K) are stored and the final best set in this list is chosen as the result.

High-dimensional space searching is achieved through a floating approach that is more effective than deterministic algorithms such as standard SFS and offers a discriminatory trade-off between the quality of optimization and the speed of computation [2].

2.3 Feature Selection

Feature Selection (FS) is an approach to pattern recognition, that chooses important variables by certain criteria. Its main objective is dimensionality reduction, which is necessary in the feature space (genes) that tends to be very large in bioinformatics, making tasks like classification and network inference difficult. The mRNA expression data, for example, contain thousands of genes (features) and typically only dozens of samples [1].

This problem is known as the 'curse of dimensionality'. According to Berisha [5], the phenomenon is related to the number of training samples necessary for satisfactory inference or classification performance, that grows exponentially with the dimension of the feature space. The solution to overcome this inconvenience is applying FS by selecting only the most relevant features [1].

FS is used to identify signature genes, or to infer a prediction network between elements (genes and proteins). This network is a Gene Regulatory Network (GRN). The inference of GRNs from expression data is modeled as a series of feature selection problems [1].

This type of data has become even more common with technologies such as microarrays, together with SAGE (Serial Analysis of Gene Expression) and RNA-Seq, which enabled the massive collection of genetic information simultaneously, but ends up creating a disparity between the amount of information and samples [2].

Two essential components for FS are the evaluation component and the search component. The first refers to how to evaluate the subsets of variables for the second, which is the procedure for selecting the variables to be analyzed [2].

2.3.1 Evaluation Component

In this component, there is the criterion function – a way to evaluate the variables or their subset. An example of this function is the Mean Conditional Entropy (MCE). Entropy is a measure of uncertainty. So, a lower entropy, brings a more cohesive information [1].

The MCE is ideal for network inference since it allows to quantify the information gain and to weight the best subset. Furthermore, it allows the penalization of unobserved instances, reducing errors due to sample scarcity, characteristic of the problems already cited above. The penalization strengthens the evaluation by considering how rare or well-observed the instances are. It is worth emphasizing that the instances, in this case, are a specific combination of the found set, which are called predictors [1].

2.3.2 Search Component

In this component, there is the search algorithm, an organized procedure to list the variables to be evaluated and subsequently determine the ideal subset of predictors. This search can be classified as optimal, where an exhaustive search is performed, going through all possibilities, or as sub-optimal, where, although it does not go through all possibilities, it presents a good cost-benefit ratio between computational cost and quality [2].

The combination, therefore, with the evaluation component, such as MCE, means performing an FS where the search returns the ideal subset that minimized the entropy, i.e., that had the best predictability for the target in question among the analyzed sets, which will depend on the search component [1].

Sequential Forward Selection (SFS) is an example of a sub-optimal and single-solution algorithm, i.e., deterministic. Because it depends on a criterion function, it is also classified as a wrapper. Its main problem is the "nesting" effect, where the selected features

are never discarded. This happens because, starting with an empty set, it finds the next best feature; once found, it attempts to find, from this set, another feature that makes the set perform better according to the criterion function [1].

The "nesting" effect naturally fails to detect an Intrinsically Multivariate Prediction (IMP), which occurs when a set of predictors strongly predicts the target, but not through individual subsets, because these will have been discarded at the start [1].

The Sequential Floating Forward Selection (SFFS) is another improvement of SFS, developed to circumvent the "nesting" effect. Its principle is: from a set of 3 predictors found by SFS, SBS is applied, which tests excluding predictors from the set if the exclusion generates a better set according to the criterion function [2].

2.4 DREAM Challenges

To offer a solid ground on which network inference approaches may be compared, the DREAM project created a special scheme of gathering and unbiasedly evaluating algorithms with the help of benchmark data sets. Application of both DREAM4 and DREAM5 data sets was made common practice to assess the accuracy and performance of Gene Regulatory Network (GRN) methods of their inference [6] [7].

2.4.1 DREAM4 Context

Previous versions, like the DREAM4 Multifactorial Networks Challenge, were concerned with the creation of in silico benchmarks. The DREAM4 challenge entailed five simulated networks (Size 100), in which 100 genes and 100 samples were present in each. These benchmark topologies were based in biological reality, on transcriptional regulatory systems of *S. cerevisiae* and *E. coli* organisms [6].

2.4.2 DREAM5 Context and Performance Metrics

The DREAM5 Network Inference Challenge was a large scale blind test whose goal was to infer gene regulatory networks by identifying the regulatory targets of transcription factors (TFs) from gene expression data [7]. It used more than 30 methods between four different datasets (one *in silico* and three *in vivo*). The result of this competition was the seminal paper "Wisdom of crowds for robust gene network inference" [7], that was mainly intended to describe the predictive quality of numerous inference algorithms [7].

The information that was given to the participants took the form of the microarray compendiums with the information about the expression of thousands of the genes under given various experimental conditions. These were genetic perturbations (e.g. knockouts, overexpression), drugs, and environmental changes. The data is presented in a way that the experimental samples (microarrays) are in rows and the genes are in columns [7].

More importantly, the evaluation following the DREAM5 challenge was purely on the quantification of predictive performance in standardized measures in the form of the Area Under the Receiver Operating Characteristic (AUROC) and the Area Under the Precision-Recall Curve (AUPR) [6]. The results published, which made community-based methods a tool of great strength in inference, did not take the most significant feature of computational performance into considerations in ranking and assessment [7].

The analysis used three major networks, which have gold standards to evaluate, which showed the scale and intricacy of the GRN inference [6]:

- **Network 1 (*in silico*):** This is a simulated network with 1643 genes, 195 regulatory genes, 805 samples and 4012 validated interactions [6].

- **Network 3 (*E. coli*):** This is the particular benchmark used in this thesis, which is based on real gene expression data of the bacterium *Escherichia coli*. It is a problematic dataset that contains 4511 genes, 334 transcription factors, 805 experimental samples, and a gold standard of 2066 validated interactions [6].

Table 2.1: Published accuracy results for top-performing methods on the DREAM5 benchmarks [7].

Method	Net 1 AUPR	Net 1 AUROC	Net 3 AUPR	Net 3 AUROC	Net 4 AUPR	Net 4 AUROC
GENIE3	0.291	0.814	0.094	0.618	0.021	0.517
TIGRESS	0.302	0.783	0.070	0.596	0.020	0.517
CLR	0.254	0.771	0.075	0.591	0.020	0.516
ARACNE	0.187	0.763	0.069	0.572	0.018	0.504
Winner (GENIE3)	0.291	0.815	0.093	0.617	0.021	0.518
2nd (ANOVerence)	0.245	0.780	0.119	0.671	0.022	0.519
3rd (TIGRESS)	0.301	0.782	0.069	0.595	0.020	0.517

Note: The table shows that it is challenging to infer well the structure of large *in vivo* networks, especially Network 3 (*E. coli*), where the best AUPR score an individual method has obtained was 0.119 [6].

- **Network 4 (*S. cerevisiae*):** The network was obtained with data on *Saccharomyces cerevisiae* consisting of 5950 genes, 333 regulatory genes, 536 samples, and 3940 verified interactions [6].

The reported accuracy of the best-performing methods on the DREAM5 benchmarks are summarized in Table 2.1, with the performance levels being moderate, in particular, on the *in vivo* networks (Network 3 and 4) compared to the *in silico* one (Network 1) [6].

2.5 The DimReduction Software

The Feature Selection Environment for Genomic Applications (*DimReduction*) open-source software was developed as a cross-platform graphical environment aimed at bioinformatics, with the goal of assisting in feature selection for genomic data [1].

DimReduction provides an interactive environment that enables the application of different feature selection algorithms, facilitating the analysis and visualization of results [1]. It was implemented in Java, offering a graphical user interface (GUI) and supporting several functionalities, including: Data Import and Preprocessing; Feature Selection; Graphical Visualization of Results; Genomic Network Inference [1].

DimReduction has two major modes of network inference: full network inference and target-specific inference. In full network inference mode, the tool conducts a thorough analysis of all the genes to form a global regulatory network, which is activated whenever no target genes are specified. Alternatively, any one or set of target genes can be specified

to conduct a targeted, localized study of regulatory interaction. This has made it very flexible to both general, search type studies and specific, hypothesis testing research.

These features make the software useful for both biological data analysis and general machine learning applications. The Appendix A provides a quick-guide on how to setup and run the original DimReduction tool with a small portion of a DREAM5 dataset.

Although *DimReduction* was a significant advancement for genomic data analysis, it has some limitations:

- It requires manual interaction to load and process data, making automation difficult.
- It lackes parallel execution capability, affecting efficiency.
- No native support for use in computing clusters or execution on headless servers.
- Although Java is a cross-platform language, nowadays the bioinformatics community prefers tools developed in Python due to the vast availability of specialized libraries such as NumPy, SciPy, and Scikit-learn. Having a Python version would increase the chances of *DimReduction* engaging with a more active community.

These limitations were the original motivations behind the development of this thesis, although some course corrections were necessary along the way (namely, the Python version turned out to be under-performing).

2.6 Related Methods

The accurate reconstruction of Gene Regulatory Networks of gene expression data has become a focus in computational biology and has been used to develop numerous dedicated computational methods. The techniques usually deal with time series or differentiation expression data, which provides information on how often genes change their expression levels. A major challenge though is that inference methods tend to be specialized to the datasets with certain behavior, i.e., there is no single method that always works better

than the others in a variety of biological networks. This difference in performance requires the development of more toned down and sturdier consensus strategies [8].

2.6.1 Combined Inference Techniques

The method suggested by GENECI (Gene Network Consensus Inference) is based on the outcome of a set of powerful inference methods published in the literature as its input to the optimization process. This group of single methods constitutes an illustration of the state-of-the-art in GRN inference because they compete well in this field. The reviewed key methods, **ARACNE**, **C3NET**, **BC3NET**, **CLR**, **GENIE3** (with its **ET** and **RF** variants), **KBOOST**, **MRNET**, **MRNETB**, and **PCIT**, are all employed on the same kind of expression data to find the same problem of GRN reconstruction [8].

2.6.2 Individual Inference Techniques

Under the individual methods of inference, the two key methods are the standardized regression analysis and the regression analysis of the interviewer evaluation scores. The resulting individual techniques used can be broadly classified in terms of the underlying principles, which are generally based on either **Mutual Information** (MI) analysis or **Regression and Feature Selection** based techniques [9] [10] [11] [12].

Mutual Information-based Approaches

These approaches involve the use of Mutual Information to estimate statistical dependence of pairs of genes, enabling linear and non-linear interactions to be detected [8] [13] [9].

ARACNE (Algorithm of the Reconstruction of Accurate Cellular Networks) uses the mutual information between pairs of candidates to determine the candidate interactions. The fundamental process entails removal of indirect dependencies by the implementation of the Data Processing Inequality (DPI). ARACNE explores every 3-way interaction and

removes the one with the smallest MI, and is aimed at reconstructing the network by removing statistically dependent interactions that can be attributed to others [8] [13].

CLR (Context Likelihood of Relatedness) is an extension of the simple Relevance Network methodology. It first computes the MI between each pair of genes and then computes a score based on the empirical distribution of these MI values, which is typically normalized by an likelihood estimate. This procedure involves comparing the MI value of a gene pair with the background distribution of MI values involving either of the genes [8] [14].

C3NET (Conservative Causal Core Network) infers the most reliable, conservative network core. It makes use of the maximization step in which a connection is drawn between two genes provided that the mutual information of the two genes is maximal in either of the two genes compared to all other genes. Such a design is conservative in nature as it gives more emphasis to avoiding false positives. C3NET is limited to making inferences as to many edges as there are genes, because it draws maximum information links [8] [9].

BC3NET (Bagging C3NET) is an ensemble algorithm that is based on C3NET and aggregated through bootstrap (bagging). It creates several bootstrap collections, uses C3NET algorithm on each of them and aggregates the networks to get fine confidence value. This ensemble approach decreases the variance of the estimates, which limits the effects of noise and outliers in the expression data [8] [15].

MRNET and MRNETB (Minimum Redundancy Network) and MRNETB (MRNET with Backward elimination) are built on the Maximum Relevance / Minimum Redundancy (MRMR) principle, which had been applied previously to feature selection. MRMR tries to identify variables (genes) with high MI with the target gene (maximum relevance) and have low MI with each other (minimum redundancy), to prevent redundancy between selected variables [12]. The forward feature selection method was first used in MRNET. The **MRNETB** was developed to address one of the known shortcomings of forward selection - that the performance is highly dependent on the starting variable selected.

MRNETB does this by substituting the forward selection with a more formidable procedure of backward elimination that is accompanied by sequential replacement [16].

Regression and Correlation based Approaches

Such techniques can be divided into techniques that reduce the network inference problem to a sequence of regression problems, or employ elaborate correlation analyses, frequently solving non-linear or combinatorial dependencies.

GENIE3 (Gene Network Inference with Ensemble of trees) is a network inference method that reformulates the network inference problem into p separate regression problems, with p the number of genes. The prediction of the expression of an individual target gene against all the input features is done with the expression of the rest of the genes (the potential regulators) as input features. GENIE3 employs feature selection using ensemble algorithms on decision trees namely **Random Forests (RF)** or **Extra-Trees (ET)**. The interactions are also given high confidence when the factor gene has a significant contribution to the expression of the target gene. This is useful in dealing with non-linear and combinatorial interactions and gives directed GRNs. Nevertheless, the efficiency of the recovery of a regulator is likely to decline with an increase in the number of regulators that influence the gene of interest [8] [10].

KBOOST is a regression model using Kernel Principal Component Analysis (PCA) and gradient boosting, and the combination of results based on Bayesian Model Averaging (BMA). Kernel PCA regression (KPCR) is a non-parametric analysis method that is helpful in estimating multiple regulation relationships. KBoost is based on a greedy search strategy that is used to sequentially choose the most impactful Transcription Factor (TF) subsets. The last probability of the regulation is determined using BMA on all the studied models. KBOOST is known to be fast in running and also has the capability to false positively detect [8] [11].

PCIT (Partial Correlation coefficient with Information Theory) is a combination of partial correlation coefficients and a criterion based on an information theory. PCIT computes three first-order partial correlation coefficients for each triplet of genes and imposes the DPI theorem to identify a local threshold (ϵ) to be used to filter the interactions. This feature of relying on a local, data-oriented threshold that is not maintained at fixed global levels as found with other methods enables PCIT to detect meaningful, though moderate, correlations detect [8] [11].

Summary of Individual Techniques

Table 2.2 sums up the technical features of the discussed GRN inference methods, addressing their key calculation procedures, filtering mechanism, and distinguishing features.

2.7 Parallelism

The increasing in processing power brought by the transition to many-core architectures is dictated by the drive towards greater performance, that is, modern algorithms demand increasing levels of computational capabilities, needing to explicitly utilize parallelism in order to achieve meaningful speedup [18], [19]. The same happens with CPU-bound computational problems, like feature selection, with at least two parallel implementations being later discussed in this thesis. Next, several core concepts related to parallelism are summarized, specifically related to the use and evaluation of parallelism in this thesis.

2.7.1 Parallelization Approaches

There are two main approaches to the decomposition of a problem, aiming to solving it in parallel. **Data parallelism** is based upon breaking down the problem domain, usually represented by data elements, into a number of independent sub-problems in which a the same shared operation is done on the whole set. This contrasts with **task parallelism** in which the problem is broken down into several functional streams applied to shared data.

Because feature selection problems may also require very similar computations to be repeated (e.g. statistical tests, distance calculations) many times, on very large data sets or subsets of features, they have the typical traits of a **data-parallel problem** [18], [19]: the workload can be partitioned into a large number homogeneous computations. This degree of parallelism can best be handled with **thread-level parallelism**

2.7.2 Thread Parallelism

The basic principle of a multi-core CPU is that it is also a MIMD (Multiple Instruction, Multiple Data) architecture, which is a replication of processing units that can execute autonomous and separate streams of control [18], [19]. The common mechanism of simultaneously working in such architectures is **thread parallelism**, in which several streams of instructions are simultaneously run on the same processing unit or node [18]–[21].

Thread-level parallelism may be used to support **fine-grained parallelism**, where the workload is partitioned into a large number of small duration homogeneous operations, or to support **coarse-grained parallelism**, where by several small grain operations are aggregated in on coarse-grain task that will be handled by a specific thread. The choice of granularity should be done in the way that better balances the number of threads involved and the time and resources required to manage their concurrent/parallel execution (i.e., too many threads may overload the system, and too few threads may underutilize it).

In this thesis, the thread parallelism exploited is of the coarse-grain type: a loop, that represents the main-hotspot in the DimReduction code, has independent iterations, being split by several threads. However, because the iterations workload is not necessarily homogeneous, different load-balancing mechanisms are used (see section 3.6).

2.7.3 Performance Evaluation

The proper model to measure the enhancement of optimizing the feature selection algorithm using the fine-grained parallelism is the **fixed-size speedup** model [21], [22].

Usually, traditional speedup (S_N) can be defined as the ratio between the sequential

execution time (T_1) and parallel execution time (T_N) when using N processors [21]–[23].

$$S_N = \frac{T_1}{T_N} \tag{2.4}$$

Parallel Efficiency (E_N) measures how well the available processing resources (N) are utilized. An efficiency of 1 is optimal usage, which is achieved at perfect linear speedup.

$$E_N = \frac{S_N}{N} = \frac{T_1}{N \cdot T_N} \leq 1 \tag{2.5}$$

Amdahl's Law [23] is linked to the fixed-size model, which supposes that a problem size is held constant while the number of processors (N) is varied [21], [22]. This method is particularly called **strong scaling**. Because the minimization of overall execution time (latency) of a given workload is the overall goal in the optimization of a feature selection process (where the input dataset size is fixed), the fixed-size speedup model is the most applicable evaluation measure [21], [23]. It quantifies the ratio of the speed with which the solution may be obtained on the parallel architecture to the optimum sequential method.

2.7.4 Scalability and Constraints

The overall maximum performance improvement brought by the parallelization of an algorithm is theoretically bounded by the portion of the sequential fraction of the algorithm [21], [22]. This restriction is measured by **Amdahl's Law**, which is a model of speedup under the assumption of the fixed size [21], [22]. The law is formulated by partitioning the workload into a parallelizable fraction (p) and an inherently sequential fraction ($1 - p$). The speedup function $S_{Amdahl}(N)$ for N processors is defined as:

$$S_{Amdahl}(N) = \frac{1}{(1-p) + \frac{p}{N}} \tag{2.6}$$

Amdahl's Law naturally (and also pessimistically) imposes a limit to the maximum speedup achievable (S_{max}) due to the unavoidable sequential component of the computation, even with an infinite number of processors ($N \to \infty$) [21], [22]:

$$S_{max} = \lim_{N \to \infty} S_{Amdahl}(N) = \frac{1}{1-p} \tag{2.7}$$

2.8 Summary

This chapter gave the theoretical background for the optimization of the DimReduction tool. It was pointed out that the inference of GRN can be formulated as a cascade of feature selection problems, intended to overcome the 'curse of dimensionality' inherent in genomic data. The method employs concepts of Information Theory with the uncertainty measure being the entropy of Shannon and Tsallis. The basic assessment module is the MCE metric, which is reduced to identify the most informative subset of features. The search part is usually based on sub-optimal heuristics like the SFFS algorithm which is an improvement over simple algorithms because it does not have the nesting effect.

The validation of the DimReductiom tool optimization attempt make us of the DREAM5 Network Inference Challenge, where the actual Escherichia coli dataset (Network 3) is employed. Even though the benchmark assessments available today consider largely the quality metrics of prediction (AUROC/AUPR), this dissertation solves the key bottleneck of computational performance. DimReduction was put in state of art by a review of related inference methods (such as Mutual Information methods such as ARACNE and CLR, and Regression/Feature Selection methods such as GENIE3 and KBoost).

Lastly, it was established the relation between the optimization of the DimReduction tool and parallelism. The feature selection has features of data parallelism and thus, multi-core CPU architecture and thread-level parallelism is the right model to be applied. The quantitative performance improvement model relies on the fixed-size speedup model, i.e. using the fixed-size speedup model and based on the metrics used including Speedup(S_N) and Parallel Efficiency(E_N) with the performance limits being defined by Amdahl's Law.

Table 2.2: Main characteristics of the individual inference methods.

Technique	Main Calculation	Filtering/Procedure	Outstanding Properties
ARACNE	Mutual Information coefficient	Statistical thresholding using Relevance Networks; then the Data Processing Inequality (DPI) property is applied afterwards	Aims to achieve high precision by removing indirect relationships [8] [13]
C3NET	Mutual information coefficient	Selects the interaction with the largest significant Mutual Information value per gene	Conservative strategy that reinforces the goal of false negative avoidance [8] [9]
BC3NET	Mutual Information coefficient	Bootstrapping is used (bagging) and C3NET is applied separately to create many networks	Consensus perspective to refine confidence values; eliminates variance in estimates [8] [15]
CLR	Mutual Information coefficient	Filtering with computing the statistical probability of MI in the context of its network	Corrected sampling errors and removed network-context false positives [8] [14]
GENIE3_ET / GENIE3_RF	Decomposition into regression subproblems	Applies Extra-Trees (ET) or Random Forests (RF) ensemble algorithm to select/rank variables	Learns more complex interactions; generates directed GRNs [8] [10]
KBOOST	Decomposition into regression subproblems	Uses Kernel PCA Regression (KPCR), gradient boosting, and Bayesian Model Averaging (BMA)	Fast execution; can do high false positive detection; uses KPCR to model non-linear relationships [8] [11]
MRNET	Mutual Information coefficient and redundancy value	Filtering with Maximum Relevance Minimum Redundancy (MRMR) using forward feature selection	Prone to drawbacks where quality highly relies on the initial variables chosen [8] [12]
MRNETB	Mutual Information coefficient and redundancy value	Redresses MRNET shortcomings by using backward elimination instead of forward selection and sequential replacement	Enhances better robustness than MRNET by examining the concept of redundancy in a more detailed manner. [8] [16]
PCIT	Partial correlation coefficients with Information Theory	Works with first-order partial correlation coefficients and applies DPI to generate a local threshold to use in filtering candidates	Uses a data-driven local threshold (ϵ); finds significant correlations where absolute values are intermediate in value [8] [17]

Chapter 3

Development of the Python Version

This chapter details the iterative process of the development and evaluation of a new implementation of the DimReduction method in Python, including comparisons with the original Java-based implementation. It begins with the creation of test datasets and the development of streamlined Java and Python command-line interface (CLI) versions. An initial performance comparison revealed a disadvantage in the Python version, which led to an attempt at parallelization. When this attempt failed unexpectedly, a deeper analysis using profiling tools was conducted to identify the root cause. This investigation uncovered a significant performance bottleneck related to garbage collection that, once solved, resulted in a stable and performant Python-based DimReduction platform.

3.1 Computational Environments

During development and evaluation, two different execution environments were used:

- **8-thread system**: desktop computer, used for development, bottleneck testing and initial parallelization tests, with an Intel Core i7-6700 Skylake CPU (4 Cores/8 Threads), 32 GB of RAM, running Ubuntu 24.04.2 LTS;
- **64-thread system**: KVM virtual machine in the IPB cluster, used for analysis of resources utilization, and large-scale performance evaluation, with an AMD EPYC 9554P vCPU (64 vCores), 64 GB of RAM, running Ubuntu 24.04.3 LTS.

3.2 Datasets

The experiments with the original DimReduction tool, and the new variants produced in this dissertation, used a dataset of E. coli from the DREAM5 Network Inference challenge, namely Network 3. The entire data set contains 4511 genes and 805 samples. For basic performance testing, validation, and debugging, two smaller subsets were created: a) a small **40-gene subset**, containing the first 40 genes (the dataset mentioned in Appendix A); b) a medium-size **400-gene subset**, containing the first 400 genes. This was done with a Python script that: i) loads tab-separated value (TSV) data; ii) removes non-informative genes (having less than three different values in the expression of all samples); iii) samples the original dataset of filtered genes in the first n (40 or 400) genes.

3.3 Java CLI Version

Before producing a Python version of the DimReduction tool, it was decided first to create a CLI variant of the original Java GUI version. The goal was to have a streamlined functional DimReduction tool, without the GUI code, that would allow to port the essential algorithms to another language, which in this case would be Python.

To produce the Java CLI version it was written a new `MainCLI` class, which reads a configuration file. This file is a simple ASCII-based text file, with parameters that correspond to those in the original GUI version, and other parameters needed in advanced workflows. An example of this file is shown in Listing C.1, in Appendix C.

After parsing the parameters in the configuration file, the relevant core methods are invoked to execute the required workflow, now without any GUI dependencies.

Both GUI and CLI Java versions support a workflow where the progress of the execution (the quantization and network inference output) is saved in CSV files. This allowed to compare, for the same input, the output of both versions (e.g., using a simple `diff` command), in order to validate the correctness of new Java CLI version. This matching was verified for both the small (40 genes) and medium size (400 genes) test datasets.

The commands used to run and check the results of the Java CLI version are shown in Listing D.1 and Listing D.2 in Appendix D. The source code of the Java CLI version is available at `https://github.com/JoaoVit0r/dimreduction`.

3.4 Python CLI Version

Next, the source code of the Java CLI version was manually translated to Python, which required substantial effort. No IA tools or other types of automatic converters were used, and the use of external libraries was avoided, making use of Python's native features.

The Python CLI version implements the same behavior of the Java CLI version, thus also depending on the same configuration file, and triggering the same actions. For the same inputs (test datasets with 40 and 400 genes), it was verified the same exact output of the Java CLI version, attesting the correctness of the new Python CLI version.

The commands used to run and check the results of the Python CLI version are shown in Listing D.3 and Listing D.2 in Appendix D. The source code of the Python CLI version is available at `https://github.com/JoaoVit0r/dimreduction`.

3.5 Comparing CLI Versions

A performance analysis was then conducted on the two CLI versions, in the **64-thread system**, to gain insight on the computational performance and resource utilization of each version. Each CLI version was tested with the small and medium-size datasets.

As the medium-size dataset requires significantly more time than the small one, the impact of any spurious load, external to the applications being tested, will be diluted, thus the data of a single execution will be sufficient to assess the typical behavior of the CLI applications. On the contrary, with the small-size dataset, executions are relatively fast, thus more sensible to externally induced transient loads, and so 3 executions are made, to trace more accurately the typical behavior of the CLI applications.

3.5.1 Monitoring Script

To conduct the analysis in a systematic and automated way, a custom monitoring script was developed. The script executes a particular DimReduction CLI (Java or Python) command repeatedly, with a cooldown interval between runs so that the system can restore itself to an idle state. It uses Dool, a resource usage statistics tool (`https://github.com/scottchiefbaker/dool`), to gather measurements in the background.

The monitoring workflow goes through the following stages:

1. **Execution and Data Collection:** for each DimReduction CLI execution:
    - The script starts a Dool process to log system metrics, which include: **CPU Usage** – time used in user space, system (kernel) space and waiting for I/O operations; **Load Average** – number of processes that are running or awaiting within the last 1, 5 and 15 minutes; **Memory Utilization** – amount of used physical memory and the size of the disk cache.
    - The CLI version is launched, with a timestamp taken just before it starts and another right after it ends, allowing to measure the duration of the execution.
2. **Cooldown Period:** The script enforces a 15 minute idle period after each execution, so that any residual load average of the system dissipates before the next run, ensuring a similar initial conditions occur consistently in each trial.
3. **Data Processing and Plotting:** After all the executions have finished, the script creates a set of SVG time-series plots of each type of metrics (CPU, Load, Memory), based on the logs collected. Two types of graphs are produced for each execution:
    - A **full timeline** chart of all the monitoring time with vertical lines representing the beginning and the end of the particular command execution.
    - A **zoomed-in timeline** graph that was only concentrated on the time frame that the command was actively executing.

The three metrics observed (CPU Usage, Load Average and Memory Utilization) are the most important due to the CPU-bound nature of the the DimReduction code, where very little IO activity takes place (constrained to the beginning and end of the program, when reading of the input datasets and writing of the final results). Some considerations are given next on the the nature and importance of the three metrics studied.

CPU utilization One of the most important parts of performance analysis is measuring the amount of computation a process requires of the system. This is made comparing the baseline CPU activity of the system with the one observed when the workload is running. The CPU usage is measured as a percentage (%) of the total processing capacity. Using analysis of CPU usage, one finds possible performance bottlenecks.

Load Average System load average is considered to be one of the basic measures, which indicate how many processes are using or requesting the system resources, in particular, the number of processes utilizing the CPU time. It is recorded after 1, 5 and 15 minutes and gives important information about the short-term performance bursts as well as long-term congestion of the system. The load average is a dimensionless number, between 0 and the number of CPU cores, representing the average number of processes in the run queue. The 1-minute load average is the most responsive measure of short time demand, being the one highlighted in the analysis bellow.

Memory Utilization The use of memory is an important factor that needs to be analyzed to provide efficiency and stability to the application and to ensure the application does not create memory leak or overconsumption leading to a reduced system performance.

3.5.2 Java CLI with 40 genes

CPU Usage

Figure 3.1 shows the evolution of CPU usage for 3 runs of the Java CLI version with 40 genes. In the 3 tests there was a steady trend in CPU usage: when the tests were not running, the base user CPU usage ranged between 0 and 1 %, with some spikes close to

2%; when a test started, this proportion always increased to a mean of 6-7% during the execution of **20 seconds**. Thus, the net CPU impact of each run was about 5-6%.

This persistently consistent behavior is verified by the individual runs. In all executions, the user CPU usage rises to a steady state of 6-7% with initial peak values of about 9%. A zoom-in view of the 1st run is provided in Figure 3.2. The other runs show a similar behavior (see Figure E.1 and Figure E.2 in Appendix E).

In summary, the Java CLI version has a consistent and moderate CPU footprint.

System Load Average

Once the execution time was under 1m (20 seconds), this analysis only focus on the 1-minute load average. This average for the execution of the Java CLI version with 40 genes is shown in Figure 3.3. During the 3 runs, the peak values were always constrained to moderate values, below 0.5. The 1st and 3rd executions generated peak loads of 0.3 and 0.34, respectively, while during the 2nd that load was near 0.5, but with an initial load close to 0.22, and the excess load was likely external to the Java CLI application.

The particular evolution of the system load during the 1st run is presented in Figure 3.4 (for the other runs, see Figure E.3 and Figure E.4) in Appendix E.

Memory Utilization

The memory usage during the three runs, as a proportion of the total 64GB of RAM available, is shown by Figure 3.5.

The memory consumption of the system was very stable during the idle intervals between the test runs, with values fluctuating between 5.0% and 5.2% of the 64 GB total RAM. The system memory usage in all three 20-second runs of the test did not differ much with this base, as can be observed in the plot of the 1st execution shown in Figure 3.6 (for the remaining runs, see Figure E.5 and Figure E.6 in Appendix E).

Although there was a minor and temporary decrease in the 5.0% and 5.2% range, this variation is regarded as small and far below the normal working conditions, over all executions. This small decrease can be probably explained by the optimization of

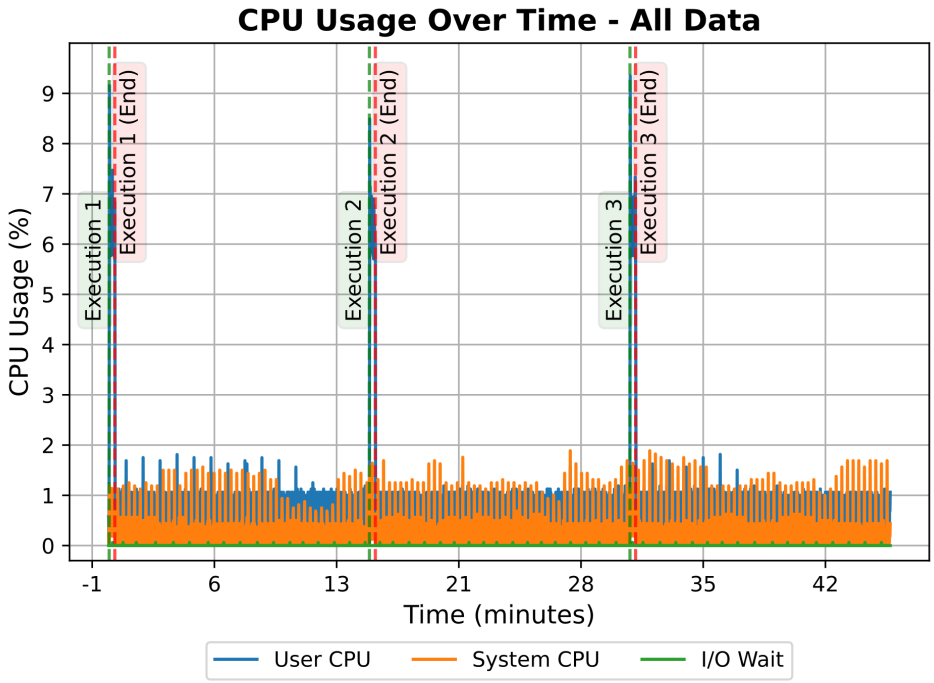


Figure 3.1: Overall CPU Usage During Three Test Runs (Java CLI with 40 Genes)

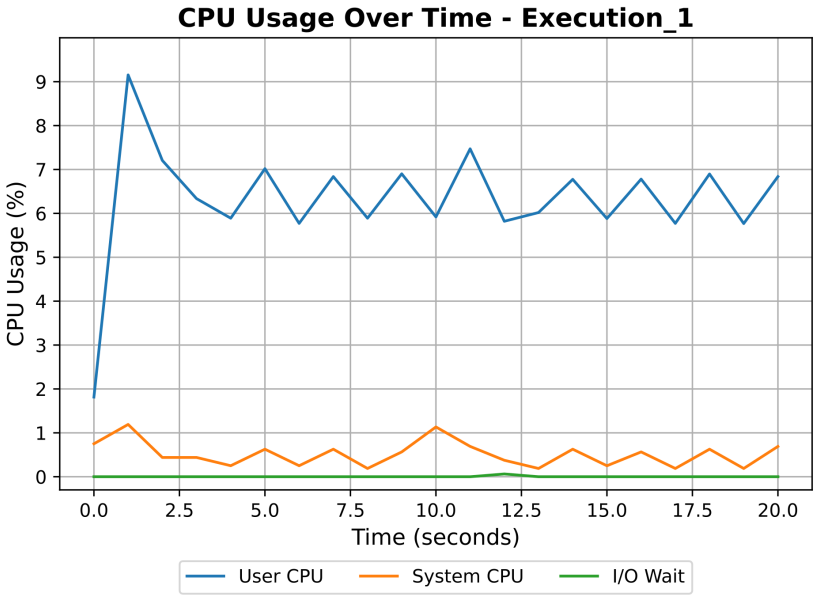


Figure 3.2: CPU Usage During Execution 1 (Java CLI with 40 Genes)

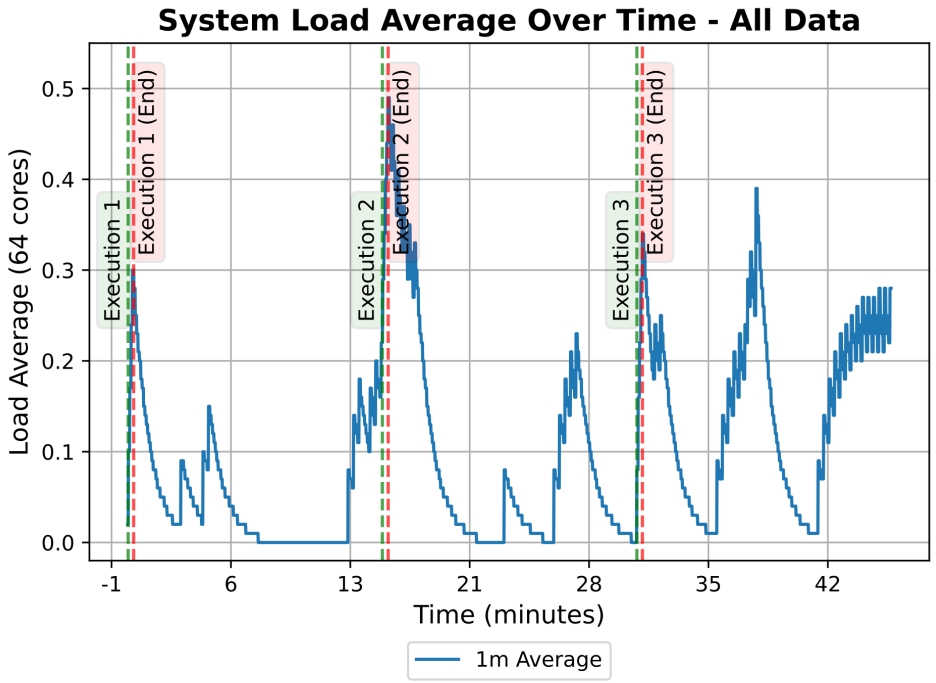


Figure 3.3: System Load Average During Three Test Runs (Java CLI with 40 Genes)

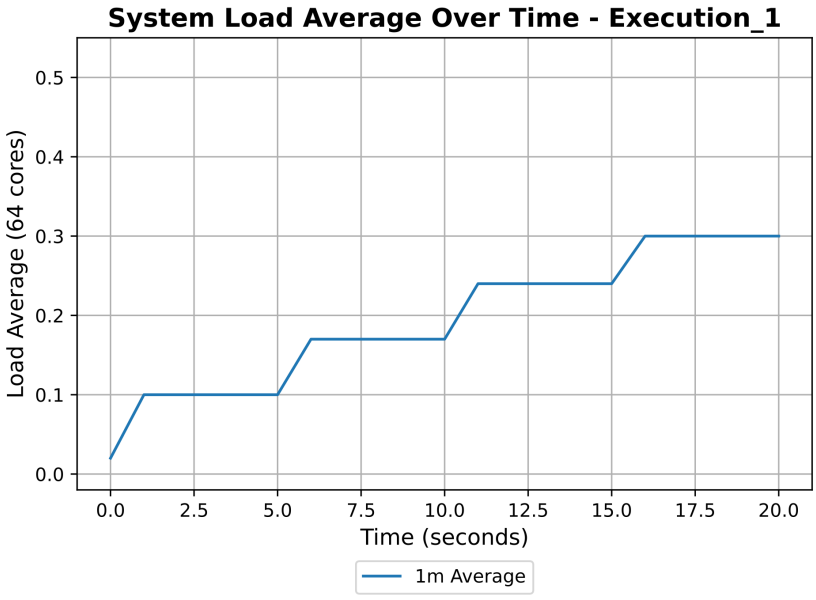


Figure 3.4: Load Average During Execution 1 (Java CLI with 40 Genes)

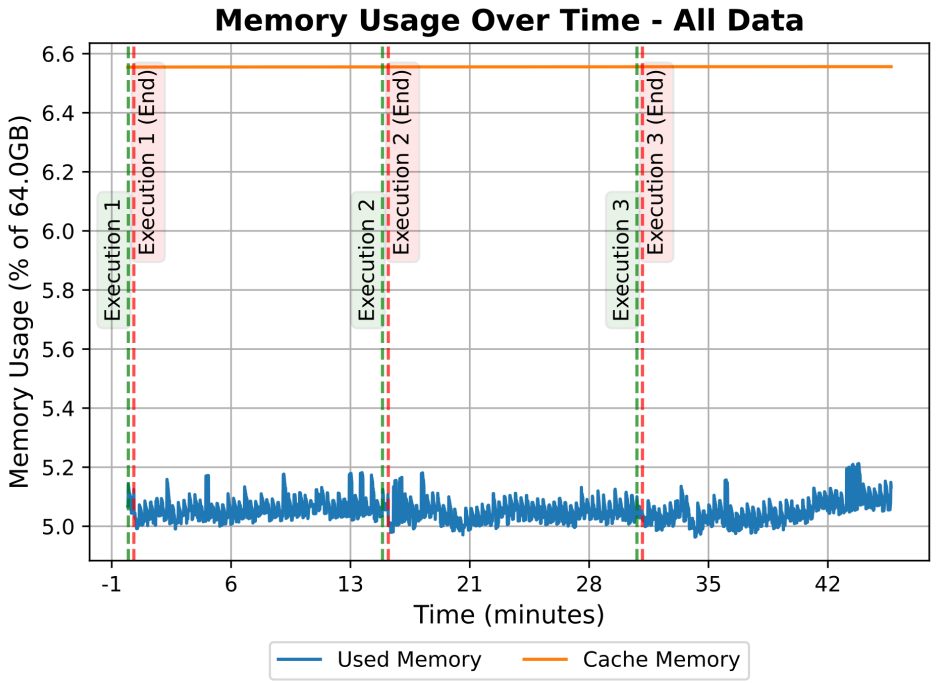


Figure 3.5: Memory Utilization During Three Test Runs (Java CLI with 40 Genes).

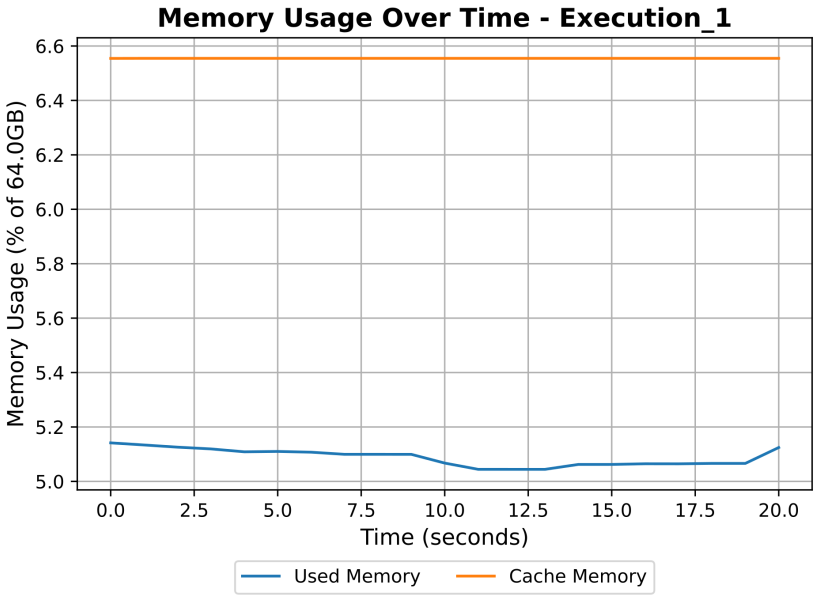


Figure 3.6: Memory Utilization During Execution 1 (Java CLI with 40 Genes)

the memory management at system level, including clearing the cache or rearranging the pages, but not by the direct activity of the JAVA CLI application itself. Thus, running this application did not result in increment in memory consumption.

3.5.3 Python CLI with 40 genes

CPU Usage

The CPU utilization of the Python CLI version with 40 genes is similar to that the Java CLI version. Again, in between the tests, the CPU usage was steady and ranged between 1 and 2 %, while it jumped to an average of 6-7% during the **14 seconds** each execution took – see Figure 3.7. Thus, the net CPU impact of the Python CLI version is about 5-6%, the same of the Java CLI version.

The evolution of the CPU usage during the 1st execution is plotted in Figure 3.8, showing an increase to an initial peak value of around 7% and an oscillation between 6% and 7% (the plots for the 2nd and 3rd run are shown in Figure E.7 and Figure E.8). This behavior is similar to the one shown by the Java CLI version in Figure 3.2, except for the initial spike, which is absent in the Python CLI version.

System Load Average

Again, once this test also takes less than 1m (14s), only the 1-minute load is discussed. As shown in Figure 3.9, the 2nd and 3rd executions generated a load under 0.3, and for the 1st execution the load was higher (near 0.5), but with an initial load higher too (near 0.22), most probably caused by system activity unrelated to the Python CLI application.

The individual plot for the 2nd run is shown in Figure 3.10, while the plots for the 1st (atypical) and the 3rd runs are shown in Appendix E (Figure E.9 and Figure E.10).

Memory Utilization

Like in the Java CLI version, when running the Python CLI application the main memory consumption was mostly unaffected, as shown in Figure 3.11.

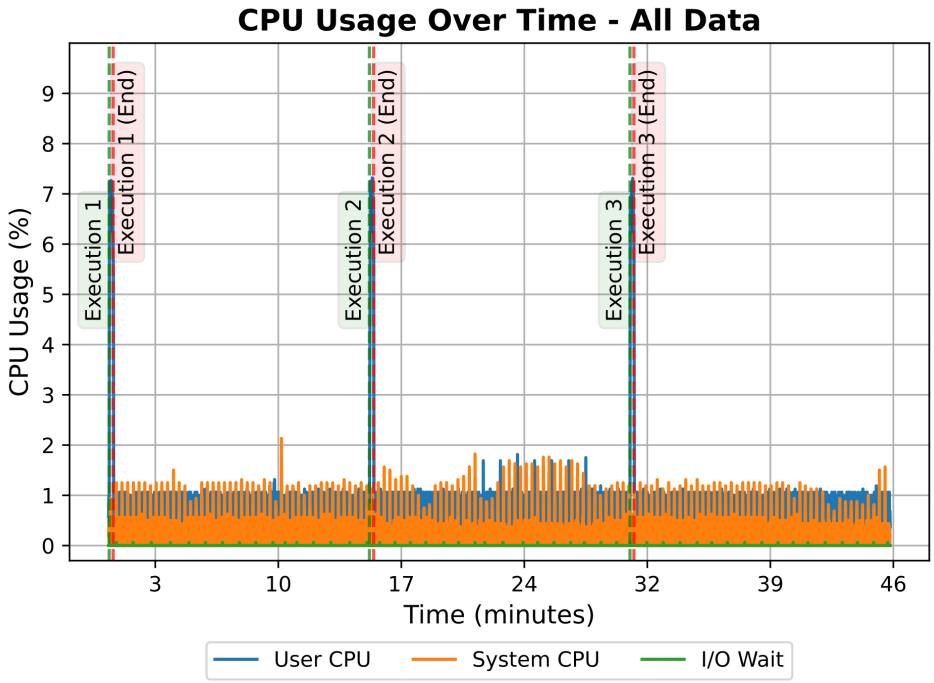


Figure 3.7: Overall CPU Usage During Three Test Runs (Python CLI with 40 Genes)

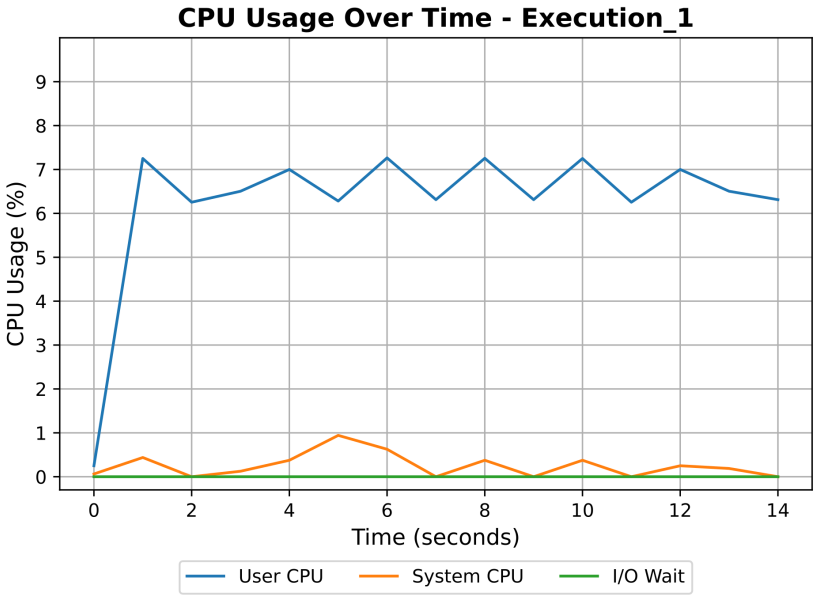


Figure 3.8: CPU Usage During Execution 1 (Python CLI with 40 Genes)

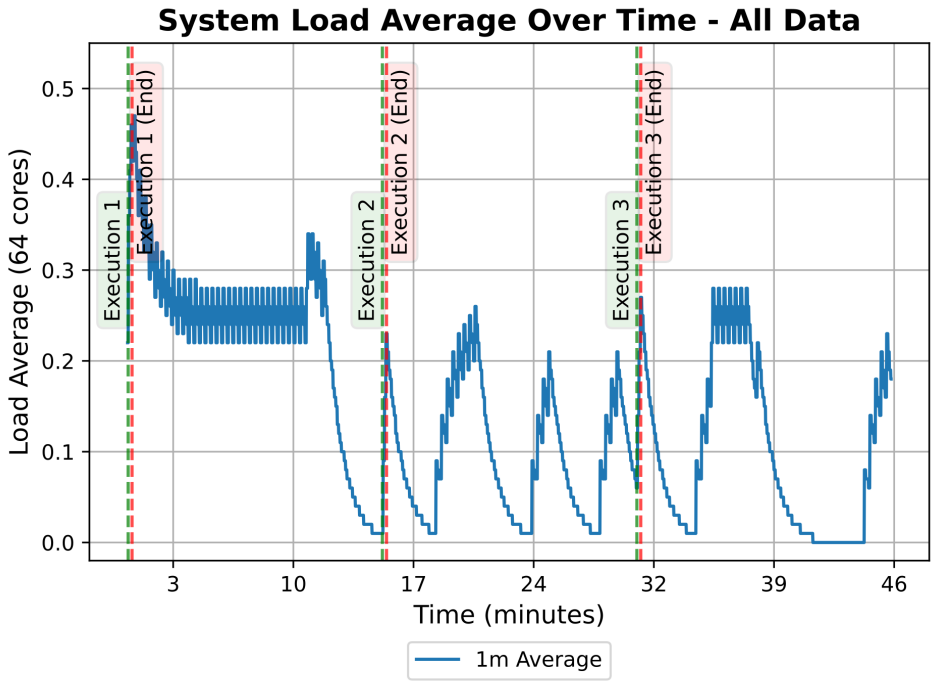


Figure 3.9: System Load Average During Three Test Runs (Python CLI with 40 Genes).

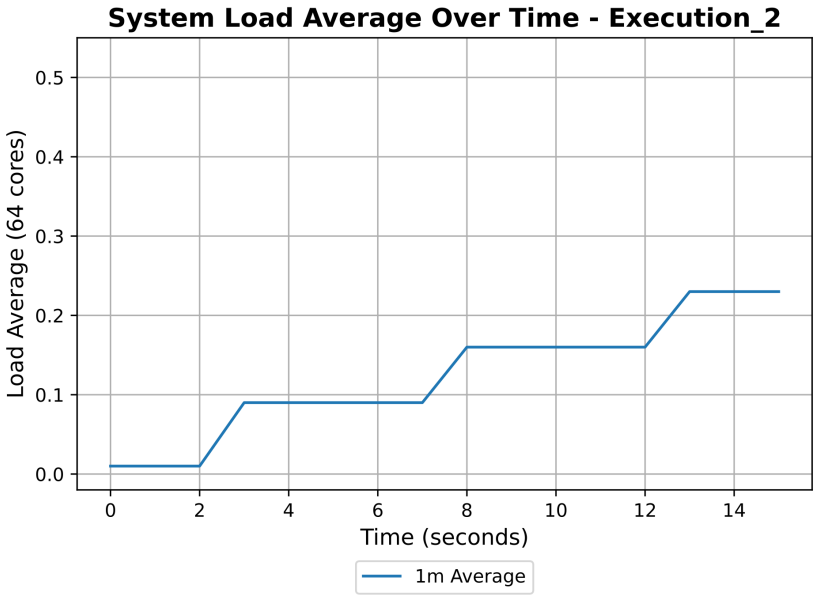


Figure 3.10: Load Average During Execution 2 (Python CLI with 40 Genes)

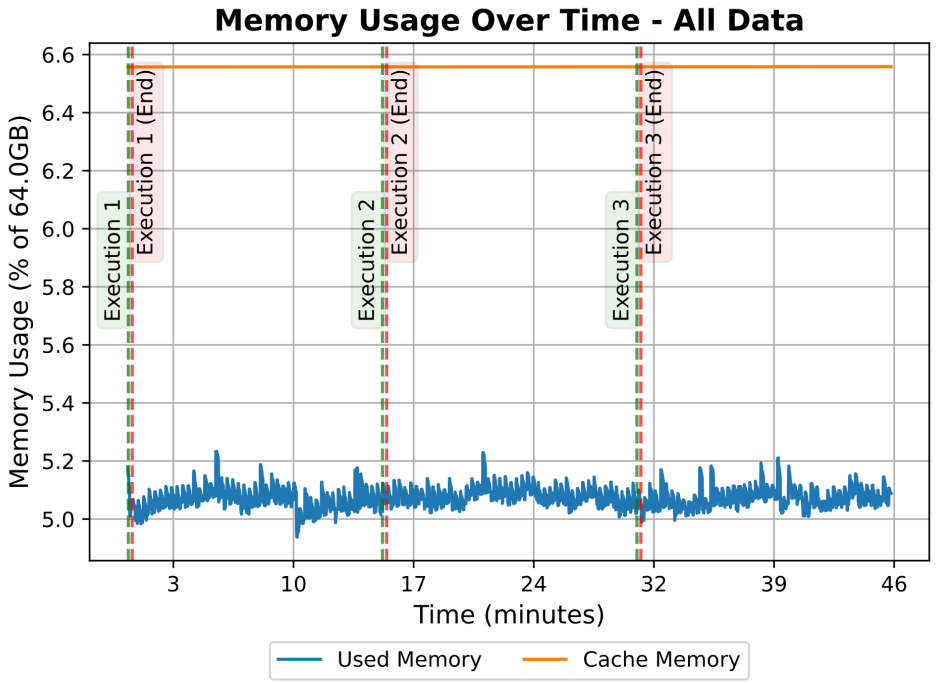


Figure 3.11: Memory Utilization During Three Test Runs (Python CLI with 40 Genes) .

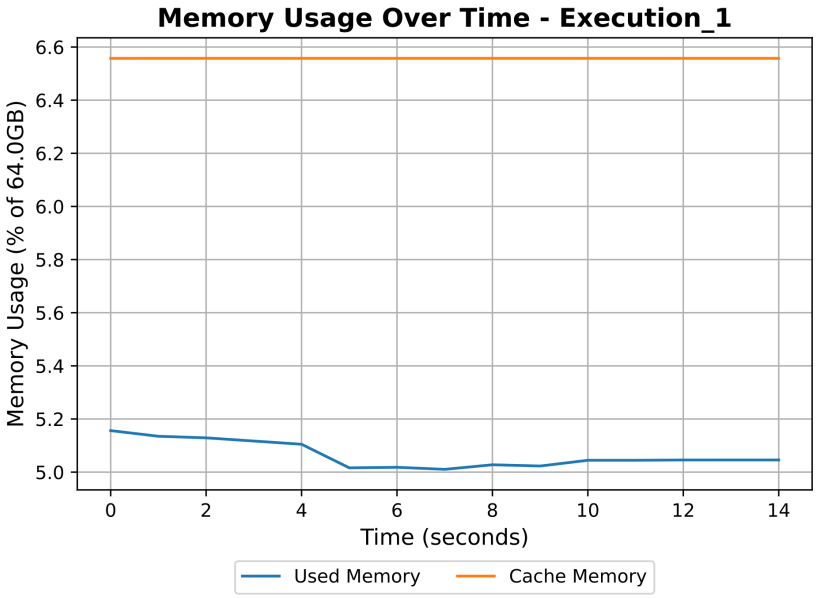


Figure 3.12: Memory Utilization During Execution 1 (Python CLI with 40 Genes)

Practically the same amount of system memory was used in all the three runs. The evolution of the memory consumed when running the 1st run is shown in Figure 3.12 (for the 2nd and 3rd executions, see Figure E.11 and Figure E.12 in Appendix E).

3.5.4 Java CLI with 400 genes

CPU Usage

Figure 3.13 shows how CPU usage evolves when running the Java CLI version with 400 genes, which took **38 minutes**. A similar baseline and execution profile can be observed in comparison with the 40 genes execution. During execution, this metric shows an initial spike of 10%, then stabilizing around 6-7%, with a few spikes, now and then, between 7% and 8%. Overall, this represents a net CPU usage of about 5-6%.

System Load Average

With an execution time above 15 minutes, the analysis of system load now considers all 3 load-average metrics (1-minute, 5-minutes, and 15-minutes). The evolution of the 3 loads when running the Java CLI version on 400 genes is presented in Figure 3.14. It can be seen that the loads were constantly limited to low values; the more reactive average (1-minute load) reached 1.2, and the medium and long-term values (5-minutes and and 10-minutes loads) converged to 1.0. This is typical of a CPU-bound serial application.

Memory Utilization

The percentage of memory consumed among the total 64GB of RAM is presented in Figure 3.15. Consumption was quite stable ranging between 5.2% and 5.4%. Thus, compared to the same test with 40 genes (recall Figure 3.5), there is a small increase of around 0.2%, which is negligible, considering the ten-fold increase in the dataset size. However, there is an increasing trend in memory consumption, which may hint at a trend typical of higher-dimensional problems, with bigger datasets.

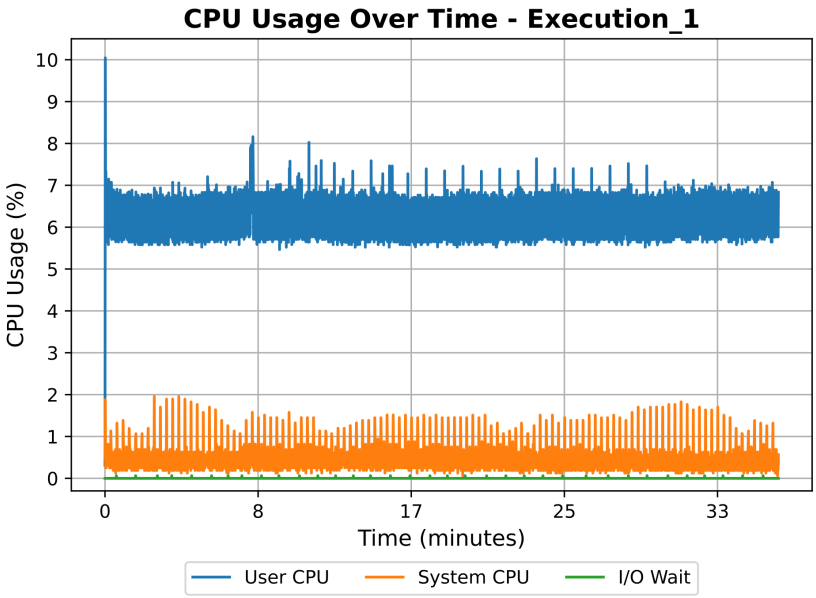


Figure 3.13: CPU Usage During Execution 1 (Java CLI with 400 Genes)

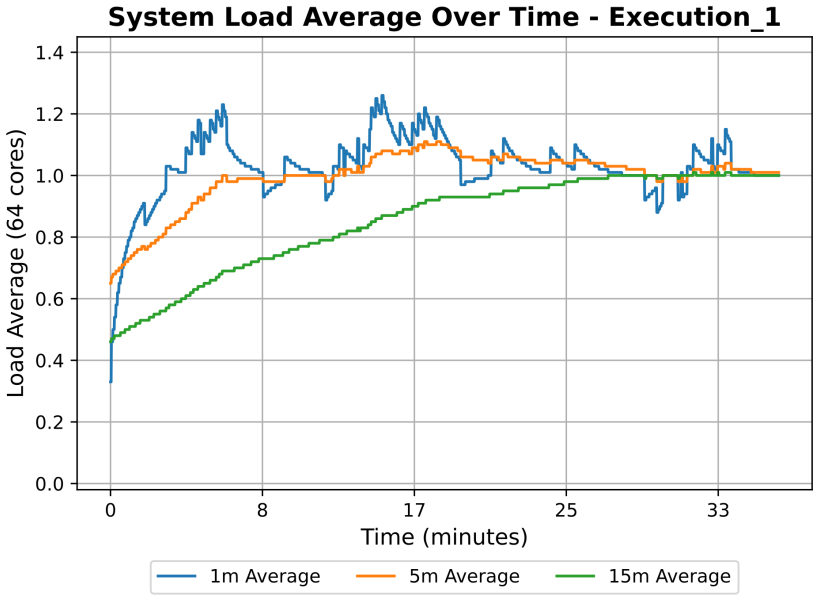


Figure 3.14: Load Average During Execution 1 (Java CLI with 400 Genes)

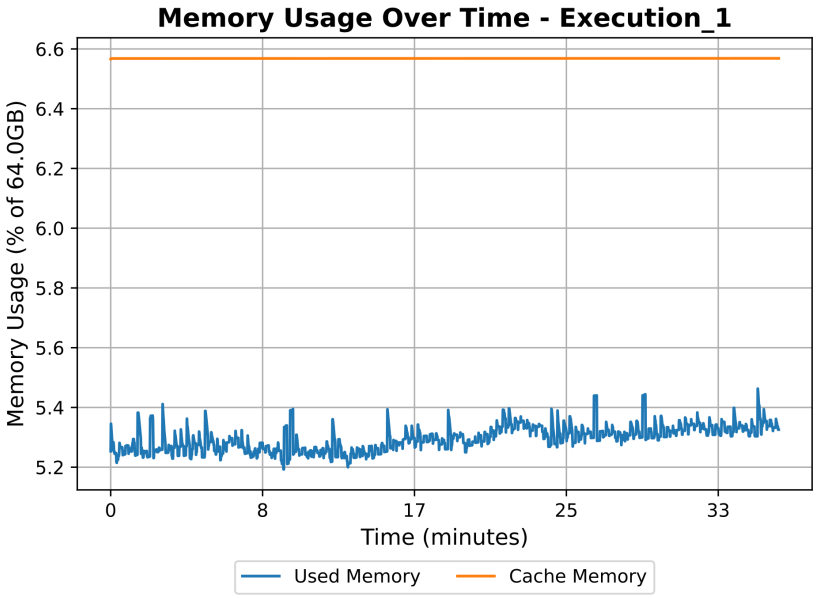


Figure 3.15: Memory Utilization During Execution 1 (Java CLI with 400 Genes)

3.5.5 Python CLI with 400 genes

CPU Usage

Similar to what was observed with the Java CLI version, using the medium size dataset makes the Python CLI version much slower than the small one, taking now **56-minute** to execute. The CPU consumption pattern, shown in Figure 3.16, is quite stable, with values slightly above Java's (recall Figure 3.13), but with absence of the initial spike that is present in the latter. Thus, the CPU consumption of the Python CLI using 400 genes shows an average of 6.2-7.2%, with the net CPU effect being within the 5.2-6.2% range.

System Load Average

Measurements of the system load during the execution of the Python CLI version with 400 genes are shown in Figure 3.17. Like in the Java implementation, the 1-minute load was mostly kept under 1.2, but there was a period in which it rose to values near 1.4; to confirm if this was exceptional, or typical, it would be necessary to repeat this test a few more times. The 5-minutes and 10-minutes loads end up showing values above 1.0, due

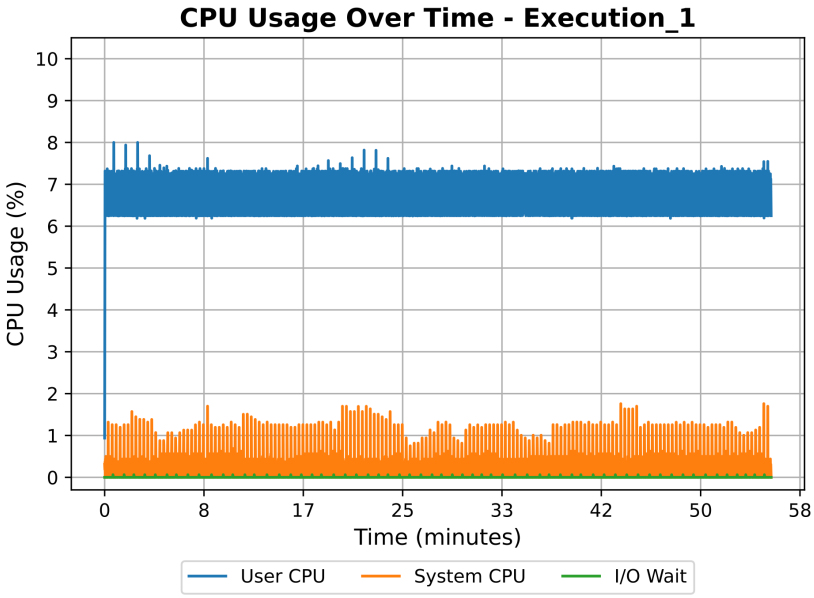


Figure 3.16: CPU Usage During Execution 1 (Python CLI with 400 Genes)

to the effect of that temporary rise of the 1-minute load, but the medium and long-term trends show an expected behavior, typical of a serial CPU-bound application.

Memory Utilization

The evolution of the memory usage of the Python CLI using 400 genes is shown in Figure 3.18. The memory consumption is kept within the same bounds (5.2 - 5.4 GB) during the entire test, with increases followed by decreases, in opposition to the consistent growing trend observed in the Java test (recall Figure 3.15). Compared with the Python test with 40 genes, where memory usage was constrained between 5.0 and 5.2 GB, the test with 400 genes consumed an average excess of 0.2 GB, which is a very small increase.

3.5.6 Summary

Looking back at the previous results, comparing the Java and Python versions in terms of resources consumption reveals a similar impact on the CPU Usage, System Load and Memory Utilization. However, this does not translate in similar execution times, neither

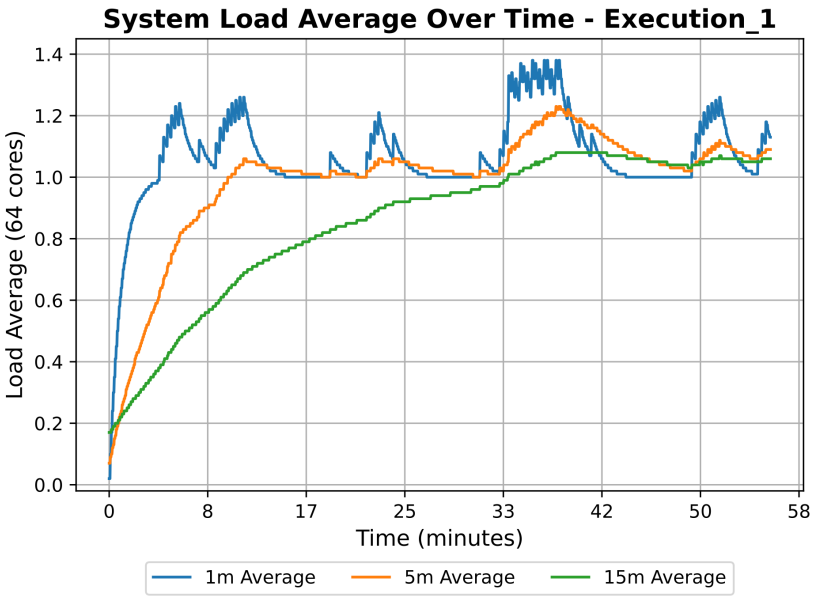


Figure 3.17: Load Average During Execution 1 (Python CLI with 400 Genes)

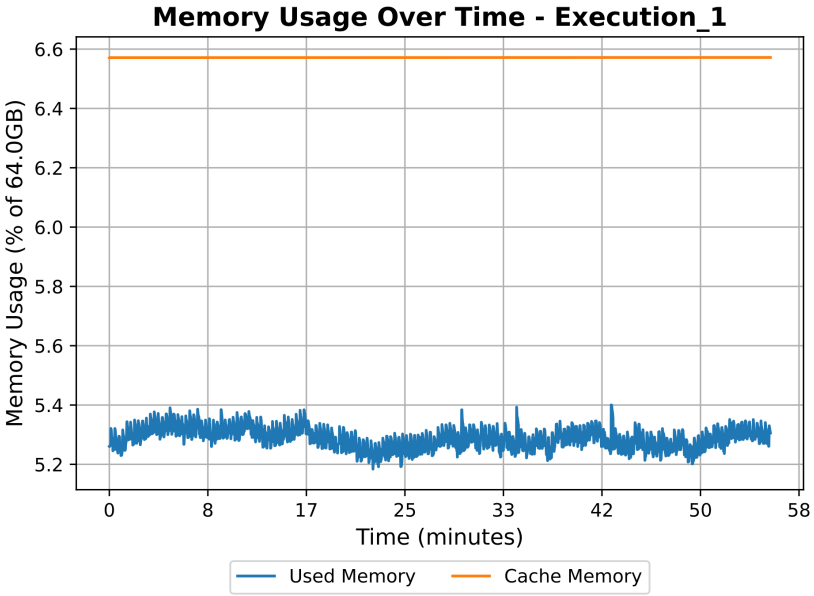


Figure 3.18: Memory Utilization During Execution 1 (Python CLI with 400 Genes)

on the trends followed by those times as a function of the input dataset sizes.

In the first comparison, with the small dataset of 40 genes, the Java CLI version was slower (execution time of 14 s) than the Python CLI version (execution time of 20 s), which seemed to validate the choice of the Python language for a more performant implementation of the DimReduction method.

However, when using the dataset of 400 genes, this ranking inverted, with the Java version taking around 38 minutes, compared to approximately 56 minutes of the Python version (a speedup of $56/34 = 1.47$). This finding meant that the Java implementation was more effective for larger, more realistic datasets, with higher computational demands.

The observation that with a more representative dataset, the Python CLI version was slower than the Java CLI version, but by a relatively small margin, led to the next rational step: an effort to make the Python version faster by exploiting parallelization.

3.6 Improving Python: Parallelization

Considering the characteristics of the Feature Selection approach (recall section 2.3) used by the DimReduction application, and the insight gained during the development of the Python CLI version, it was known that the main hot-spot of DimReduction is the network inference stage. The latter involves iterating over all the target genes, a process without inter-dependencies between any iterations, and thus easily parallelizable.

Nonetheless, one important obstacle to consider was the Python Global Interpreter Lock (GIL), which traditionally disallows the simultaneous execution of Python code by multiple threads, and is active in Python 3.12, the latest stable Python version available at the time of this work, and the one used by default. Thus, to evaluate the parallel Python version, it was used a version of Python 3.13 in which the GIL can be turned off – this GIL-free version, also known as free-threaded Python, corresponds to the Python Enhancement Proposals (PEP) 703 (see `https://peps.python.org/pep-0703/`).

To parallelize the network inference main process, that finds the predictors set for each target one by one (sequentially), there are three possible strategies (2 static, 1 dynamic):

- *Slice Groups*: The targets are divided equally among the specified number of threads. Each set is filled before the threads start, and the targets are kept in the same order they were provided. Example: For 6 targets and 2 threads, the first thread handles targets 1, 2, and 3, while the second thread handles 4, 5, and 6.

- *Stride Groups*: The targets are also divided equally among the specified number of threads, but here they are assigned in strides equal to the number of threads. Example: For 6 targets and 2 threads, the first thread handles targets 1, 3, and 5, while the second thread handles 2, 4, and 6.

- *Dynamic Assignment*: A synchronized index is shared between the threads. Each thread retrieves the index of the next available target, processes it, and then retrieves the next one until there are no more targets.

Initially, only the *Slice Groups* strategy was implemented, with the number of threads specified by a parameter. The original Python CLI *Serial* version and the new *Slice Groups* parallel version were then executed in the **8-thread system**, using the 400 genes dataset and the GIL-free Python 3.13. The results of these benchmarks are in Table 3.1.

Table 3.1: Python 3.13 CLI Serial vs Slice Groups with the 400 genes dataset

Implementation	**Threads**	**Mean Duration (h:mm:ss)**
Serial	1	2:22:47
Slice Groups	1	2:19:12
Slice Groups	4	2:25:00

As is visible in the table, the parallel version was not performing as expected, with the outcome being counter-intuitive, as the parallelization of the code not only didn't reduce the execution time but, on the contrary, worsened it.

This utterly surprising performance deterioration was a strong indication that there was a critical underlying bottleneck which was aggravated by multithreading (perhaps contention to a common resource or excessive overheads due to thread management).

3.7 Profiling Analysis

Since the parallel Python CLI implementation was acting unexpectedly, a bottleneck analysis of both the Java and Python CLI sequential versions was carried out, to determine the cause of the performance deterioration and help defining possible solutions.

The profiling data was recorded by the *perf* tool, and the flame graph visualizer of the *Hotspot* tool (available at `https://github.com/KDAB/hotspot`) was used to help analyzing the data. The settings that are needed to utilize these tools are outlined at Appendix F. Both the 40 genes and the 400 genes scenarios were examined (as the different dataset sizes may highlight different bottlenecks), and all tests were performed in the **8-thread system** (the same where the 1st parallel Python version was evaluated).

A common trend observable in the following sections is that the execution time with profiling is always larger than the execution times reported in section 3.5. This is expected, and due to the impact of the profiling mechanism. Therefore, the execution times with profiling are only provided for completeness, having no other purpose whatsoever.

3.7.1 Java CLI Performance Profile - 40 Genes

As shown in Figure 3.19, *perf* captured several interesting insights, including the utilization of 29 threads during execution and 48 seconds of running time while profiling.

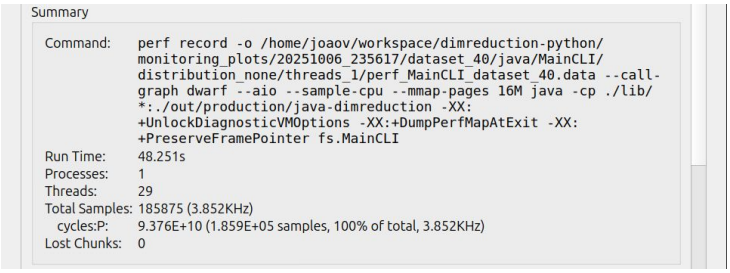


Figure 3.19: Perf summary of the Java CLI with 40 Genes

The analysis of *Hotspot*'s flame graph, shown in Figure 3.20, revealed significant Garbage Collection (GC) overhead, with GC-related costs accounting for approximately 75.5% of the total execution time. Examining the caller tree, as detailed in Figure B.1 to Figure B.3 in Appendix B, provided further evidence of the substantial GC impact.

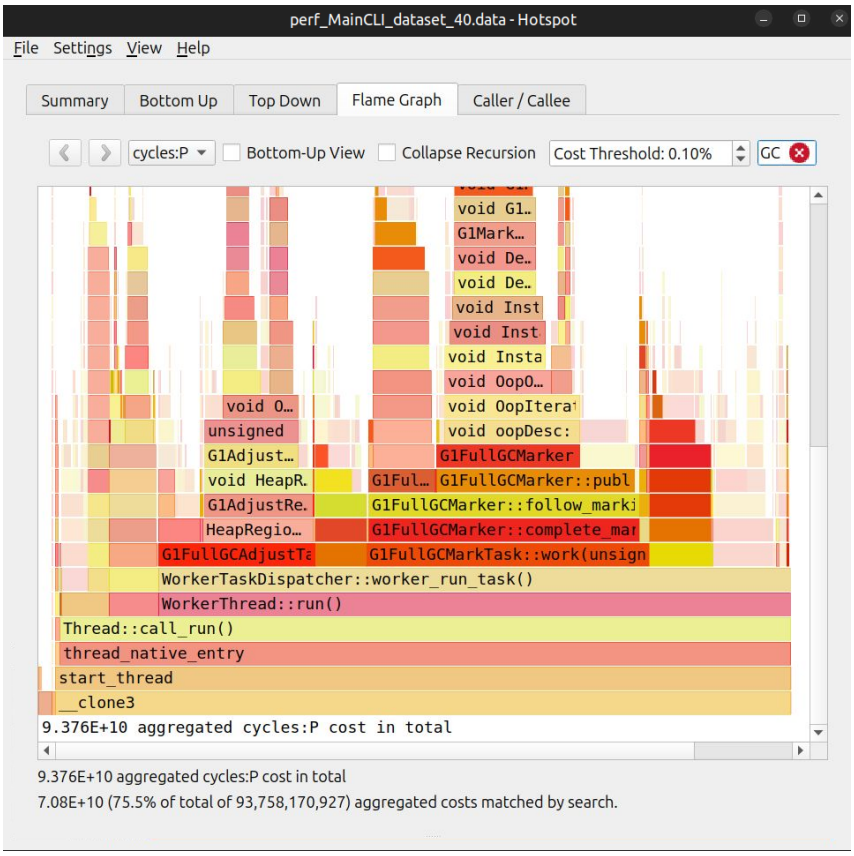


Figure 3.20: Hotspot Flame Graph of the Java CLI with 40 Genes

Taking a look at the process timeline, there are several threads related to GC operations running during the entire execution, as shown in Figure B.4 in Appendix B. Such threads include worker threads like `GC Thread #0`, which are part of the Java Virtual Machine (JVM) and are used to do parallel garbage collection. They are used repeatedly, which usually means that the application is creating a large number of temporary objects, thus causing a high rate of memory clean-ups and a large amount of computation.

Focusing specifically on the Java DimReduction code, it was found that the `MCE_COD` function execution accounted for 2.22% of the total execution time, as shown in Figure 3.21. This function computes the Mean Conditional Entropy (MCE) (recall section 2.2), which is the fundamental metric to compare subsets of features. The repetitive nature of its calculation by the search algorithm should render it an important performance bottleneck, which doesn't seem to be the case, considering the weight of the GC operations.

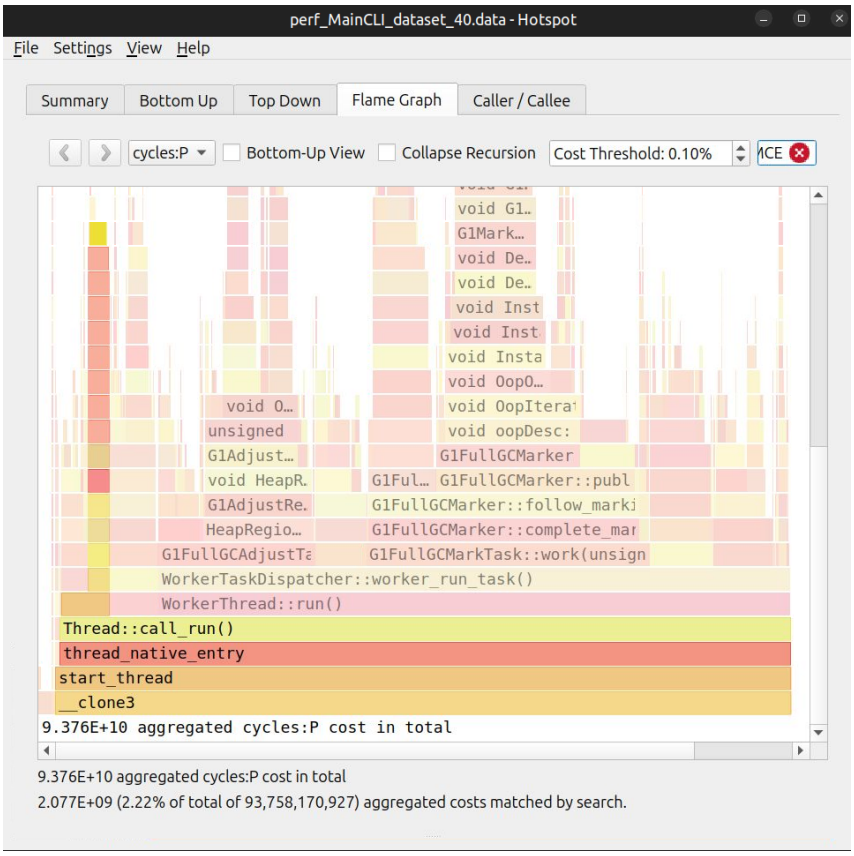


Figure 3.21: `MCE_COD` function presence in the Hotspot Flame Graph of the Java CLI with 40 Genes

To focus more specifically on the DimReduction code, the GC-related threads were filtered out from the analysis. This way, other constraints, possibly related to the nature of DimReduction algorithm, will become more prominent. The resulting filtered view can be seen in Figure B.5 in Appendix B. This filtering allowed to better isolate and examine the computational costs directly attributable to the DimReduction application logic rather than memory management overhead.

As visible in the flame graph of Figure 3.22, the filtered perspective reveals the `MCE_COD` function now representing 14.8% of the execution time, providing a more accurate view of the DimReduction implementation computational behavior. Thus, if it was possible to turn off the Java GC mechanism, or minimize its usage, the execution profile of the DimReduction application could change in a way that would make parallelization pay off.

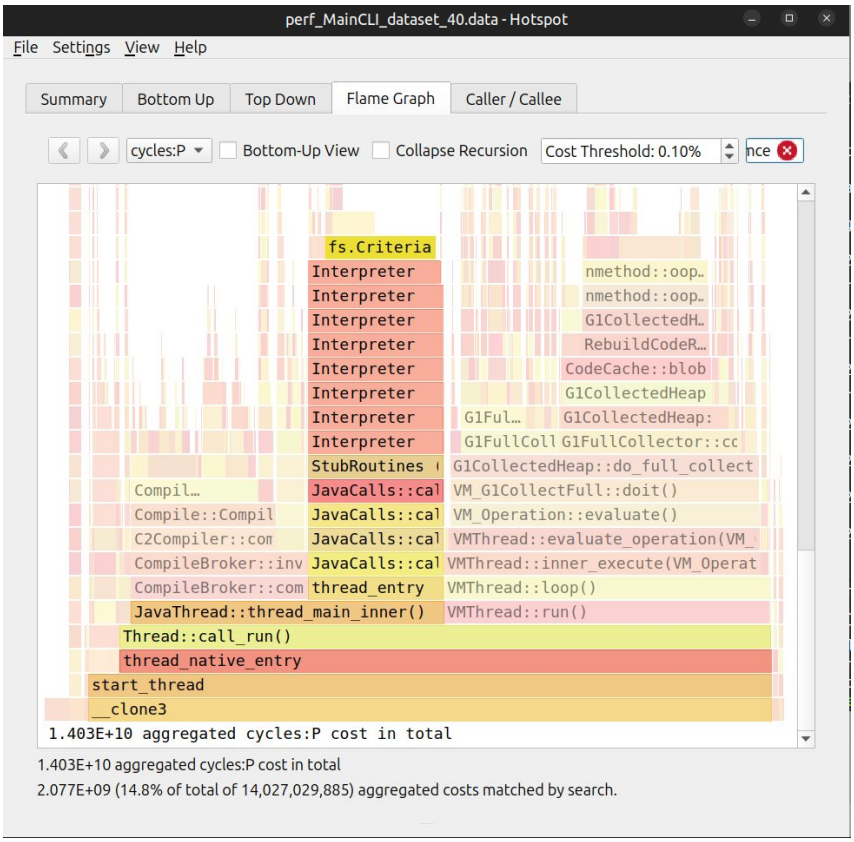


Figure 3.22: `MCE_COD` Cost for Code-Focused Analysis of Java CLI with 40 Genes

3.7.2 Python CLI Performance Profile - 40 Genes

The profiling of the Python CLI version with 40 genes revealed notable differences from the Java CLI profiling in the same conditions, including the utilization of only 1 thread and 28 seconds of profiling time, as shown in Figure 3.23 and Figure 3.24. Specifically, the latter shows no significant background threads, other than the Python interpreter.

Considering these differences, the flame graph and caller tree revealed more visible code operations, making it easier to observe the impact of `MCE_COD` in the execution. In the caller tree, shown in Figure B.6 in Appendix B, this function accounts for 94.6% of execution time. This behavior is also visible in the flame graph of Figure 3.25, which shows a single large column leading up to the function., as demonstrated in Figure 3.25.

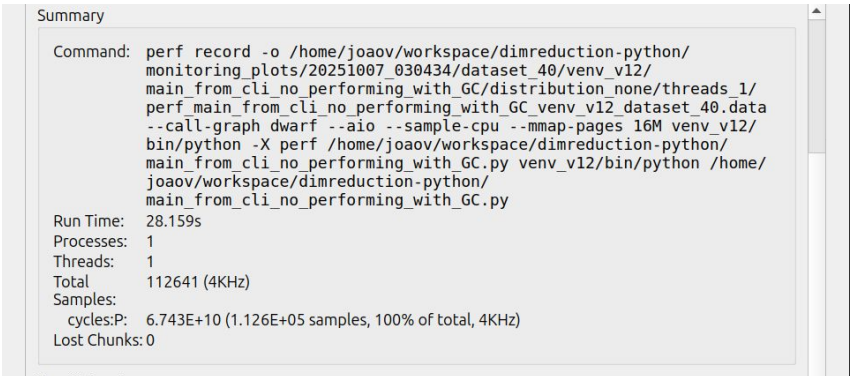


Figure 3.23: Perf summary of the Python CLI with 40 Genes

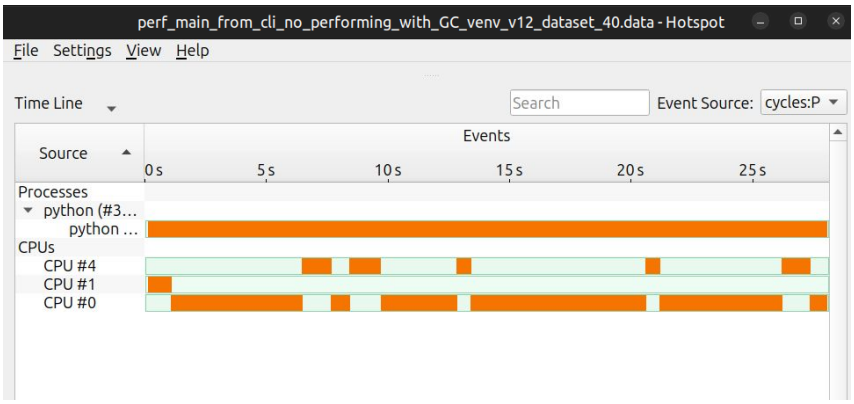


Figure 3.24: Process Timeline of the Python CLI with 40 Genes

3.7.3 Java CLI Performance Profile - 400 Genes

With 400 genes, the Java CLI implementation required considerably more time to be profiled (54 minutes), as expected. Notably, as shown in Figure 3.26, the *perf* tool registered the same utilization of 29 threads, already observed with 40 genes. The analysis of the flame graph of Figure 3.27 also reveals a significant garbage collection overhead: 78.1% of the total execution time, close to the value with 40 genes (75.5%).

Examining the caller tree, shown in Figure B.7 in Appendix B, provides further evidence of the substantial GC impact. Focusing specifically on the DimReduction code, the `MCE_COD` execution has a very low impact, accounting only for 2.22% of the total execution time, the same proportion already observed with 40 genes (see Figure B.2). The cost of the `MCE_COD` function within the flame graph in Figure 3.28 (2.14%), confirms its

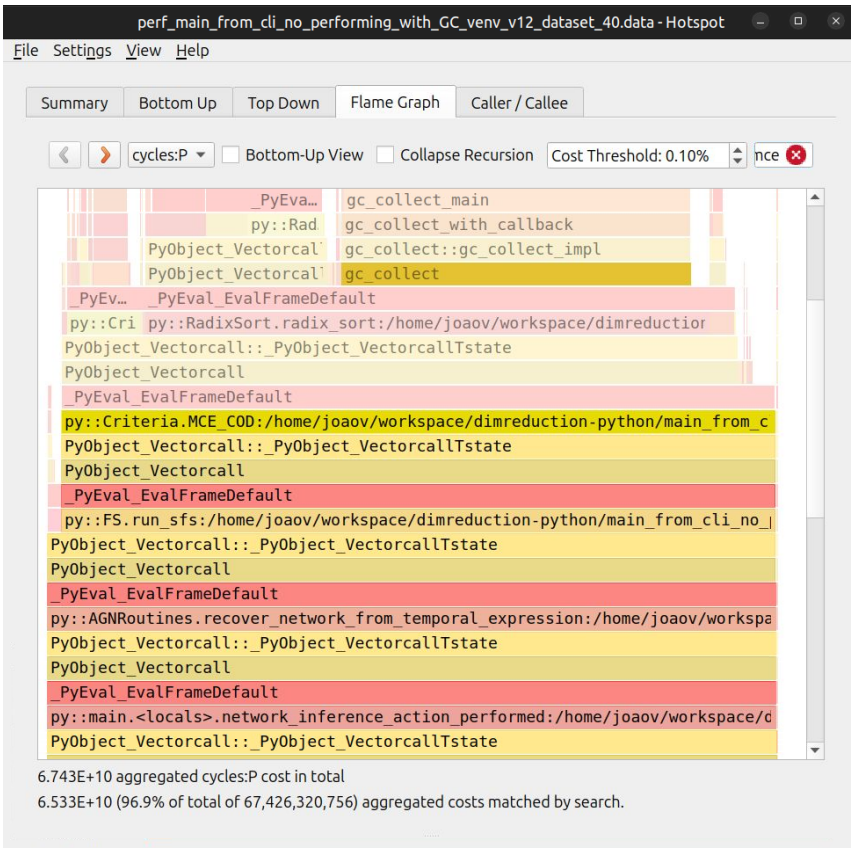


Figure 3.25: Flame Graph Showing `MCE_COD` Cost of Python CLI with 40 Genes

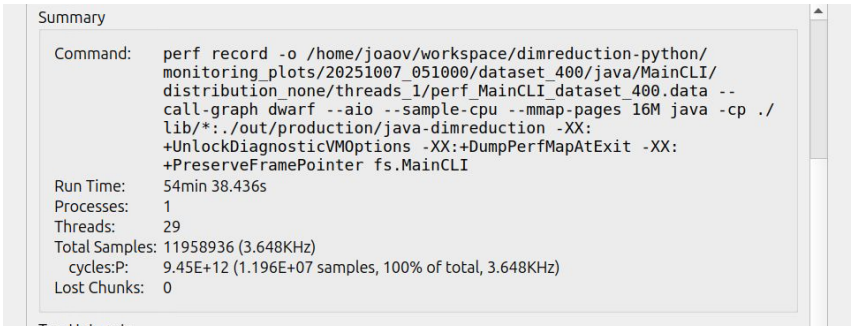


Figure 3.26: Perf summary of the Java CLI with 400 Genes

insignificance in the overall performance profile, due to the huge impact of GC activity.

The significant garbage collection overhead is also visible in the process timeline, show in Figure B.10 in Appendix B, where there are GC-related threads running during the entire execution, similarly to what was observed with 40 genes.

Filtering out the GC threads from the analysis, as shown in Figure B.11 in Appendix B,

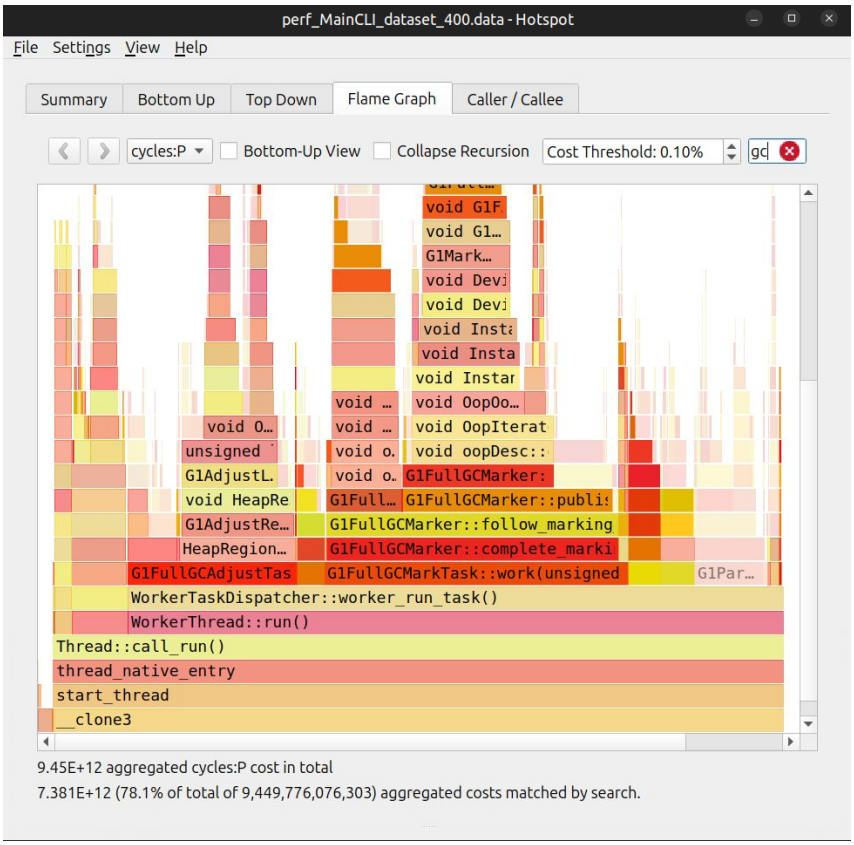


Figure 3.27: Hotspot Flame Graph of the Java CLI with 400 Genes

allows to isolate and examine the computational costs directly attributable to the DimReduction algorithms. Under this perspective, the `MCE_COD` function now represents ≈21% of the execution time, as shown in Figure 3.29. This value is above the 14% observed with 40 genes, a workload increase that is expected from increasing the number of genes.

3.7.4 Python Performance Profile - 400 Genes

The profiling of the Python CLI version analysis with 400 genes was ultimately skipped due to several problems that were not possible to solve in due time: with 400 genes, the profiling took almost double the time (≈ 1 hour and 45 minutes) of the profiling of the Java version, in line with the behaviour already observed in section 3.5, whereby the Java version was clearly faster than the Python one with 400 genes; also, the long profile sessions produced huge amounts of data (more than 120 GBytes per session), making its

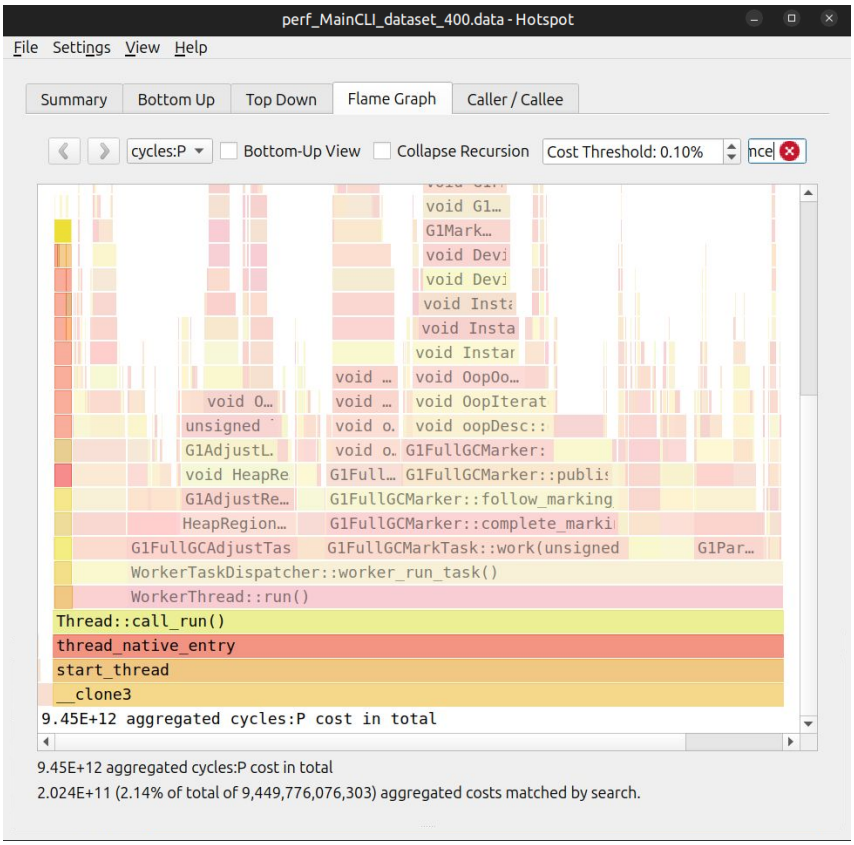


Figure 3.28: `MCE_COD` function presence in the Hotspot Flame Graph of the Java CLI with 400 Genes

analysis by the Hotpots tool impossible (even in the 64-thread machine).

Therefore, based on the comprehensive data gathered up to this point, it was assumed that the results of the 40 gene analysis with both implementations, and of the 400 gene analysis with the Java implementation, provided sufficient insight to extrapolate the expected behavior for the Python implementation with the 400 gene dataset, namely a similar relevance ($> 90\%$) of the execution time for the `MCE_COD` function.

Furthermore, the primary objective of this analysis was to identify areas where development efforts should be concentrated for performance optimization. The previous analysis with Java and Python implementations using the 40 gene dataset provided more than adequate evidence that both versions exhibited similar primary computational costs (if ignoring the impact of the Java GC), particularly dominated by the `MCE_COD` function.

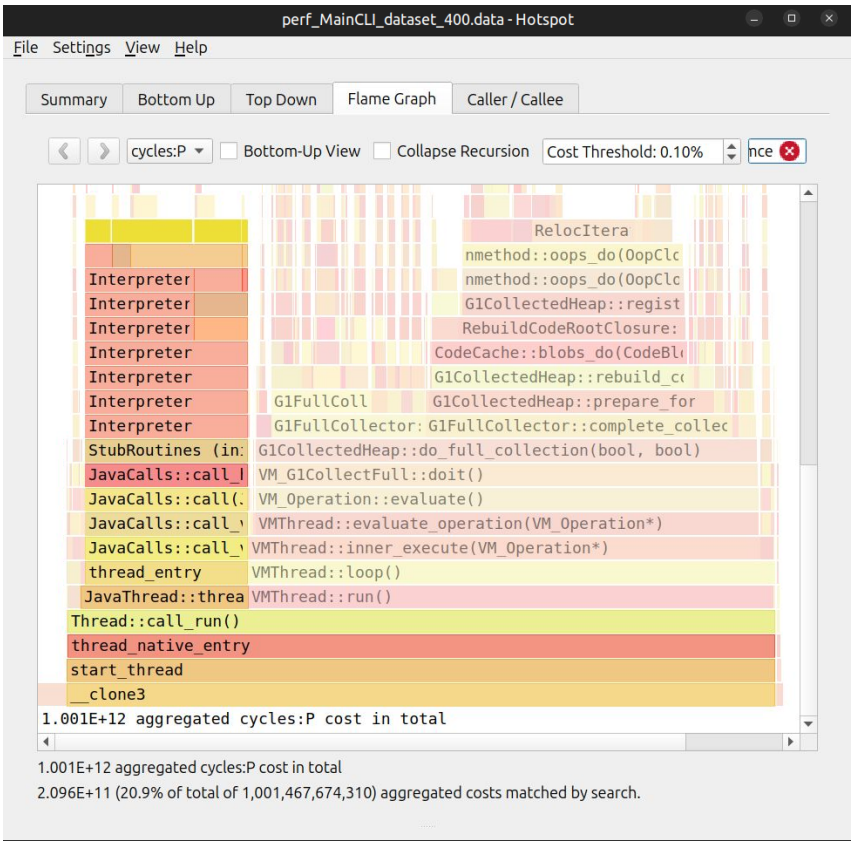


Figure 3.29: `MCE_COD` Cost for Code-Focused Analysis of Java CLI with 400 Genes

3.7.5 Summary

Two different behaviors were observed during the profiling of the Java and Python implementations of the DimReduction method. The 1st was the dissimilarity of the impact of the memory Garbage Collection. Both languages apply this mechanism, but this was much more evident in Java, where several GC background threads were always present and active. On the contrary, the Python GC activity was not noticeable. Despite this, the Java execution times were noticeable better for larger datasets, meaning that the impact of GC is counterbalanced by the highest intrinsic performance of the Java runtime.

The 2nd different behavior derives from the 1st, whereby the relative weight of the `MCE_COD` function – part of the main DimReduction hotspot – may be seen as negligible, or dominant, given the different impact of the language specific GC mechanism. The different computational costs of the `MCE_COD` function, for all scenarios previously profiled, are

shown in Table 3.2. Arguably, it can be said that the costs for the Filtered GC scenarios are probably a more realistic indicator of the true computational weight of the `MCE_COD` function, once the impact of GC was filtered out and the same effect could indeed be achieved by turning off the Java GC mechanism[1] or minimizing its usage.

Table 3.2: MCE_COD Execution Cost Summary

Language	**Num. of Genes**	**Profiling Scenario**	**MCE_COD Cost (%)**
Java	40	Default	2.22
Java	40	Filtered GC	**14.8**
Python	40	Default	94.6
Java	400	Default	2.22
Java	400	Filtered GC	**21.0**
Python	400	Skipped	-

3.8 Improving Python: Avoiding Explicit GC

Following the previous profiling, an in-depth investigation of the Java CLI code, and of the Python CLI code resulting from its translation, uncovered an explicit call, on both versions, of the GC mechanism. This was regarded as one possible reason (perhaps the main one) for the Python CLI version not achieving a performance comparable to Java's.

To investigate this hypothesis, tests were made in the **8-thread system**, with four different versions of the Python CLI code:

- **Serial-explicit-GC:** 1st version converted from Java, with the explicit GC call;
- **Serial-no-explicit-GC:** Serial-explicit-GC with the explicit GC call removed;
- **Slice-Groups-explicit-GC:** 1st parallel version, based in the Slice Groups approach, also with the explicit GC call;
- **Slice-Groups-no-explicit-GC:** Slice-Groups-explicit-GC with the explicit GC call removed.

[1] See `https://stackoverflow.com/questions/2927396/how-can-i-disable-java-garbage-collector`.

As expected, the removal of the GC explicit call didn't affect the algorithmic behaviour, with all versions still generating the same outputs for the same inputs. However, it reduced significantly the execution time by approximately 3-4x, as shown in Table 3.3, where the time scale dropped from almost 2 and a half hours to less than 40 minutes.

Table 3.3: Performance Impact of Garbage Collection on Python CLI with 400 genes

Python CLI version	**Duration (h:mm:ss)**	**Speedup**
Serial-explicit-GC	2:22:47	—
Serial-no-explicit-GC	0:38:15	3.73
Slice-Groups-explicit-GC (1 thread)	2:19:12	—
Slice-Groups-no-explicit-GC (1 thread)	0:38:02	3.66

In addition, using more than 1 thread now clearly pays off, as shown in Table 3.4, with parallel efficiencies >86% for up to 4 threads (with 8 threads, the **8-threads system** is showing the limitations of hyper-threading, once the CPU has only 4 physical cores).

Table 3.4: Slice-Groups-no-explicit-GC Thread Scaling Performance, with 400 genes

Number of Threads	**Duration (h:mm:ss)**	**Speedup**	**Efficiency (%)**
1	0:38:02	—	—
2	0:20:25	1.86	93.17%
4	0:11:02	3.45	86.23%
8	0:10:29	3.63	45.38%

This result, showing the initial parallelization effort finally yielded the expected performance uplift, lead to try domain division approaches yet to be tested at this stage: *Stride Groups* and *Dynamic Assignment*. These were implemented and evaluated, and the results are show in Figure 3.30, together with the results of the *Slice Groups* approach.

The *Dynamic Assignment* performance matches *Slice Groups*'s, but the *Stride Groups* ensures a performance improvement, although rather modest, as visible in Table 3.5.

After this analysis, and as a result of the elimination of the explicit GC call in the original code, it can be concluded that the multithreaded version of the Python CLI implementation of DimReduction can be used as an effective alternative to the Java-based implementation, specially by those from the BioInformatics community that may

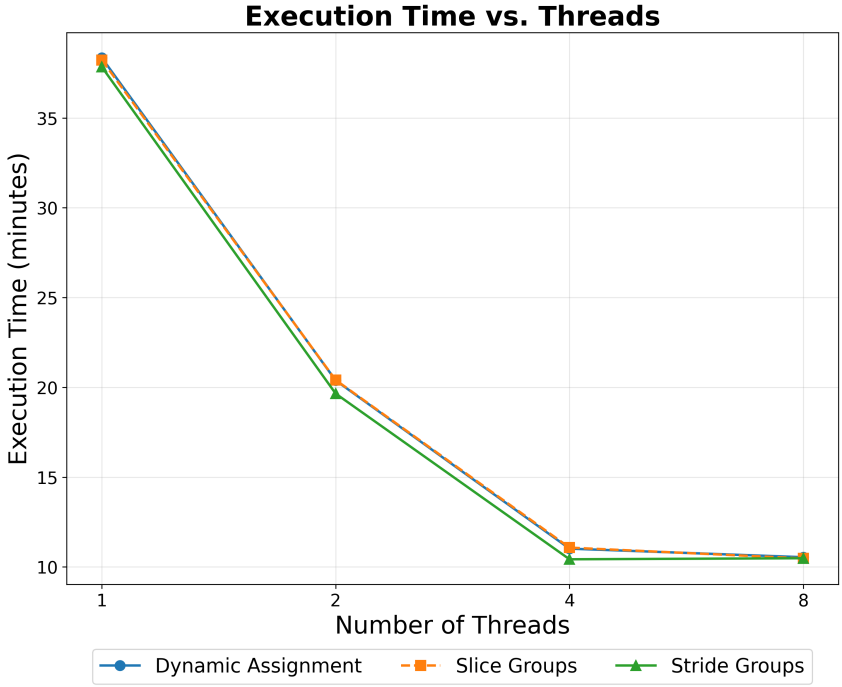


Figure 3.30: Python Execution Time Relative to the Number of Threads on the 8-core machine (400-gene dataset).

Table 3.5: Stride-Groups-no-explicit-GC Thread Scaling Performance, with 400 genes

Number of Threads	**Duration (h:mm:ss)**	**Speedup**	**Efficiency** (%)	**Speedup relative to Slice-Groups-no-explicit-GC**
1	0:37:51	—	—	1.00
2	0:19:39	1.93	96.29%	1.03
4	0:10:26	3.63	90.75%	1.05
8	0:10:29	3.61	45.12%	0.99

be interested in its integration with other Python-based codes. However, it remains to be studied the effect of the removal of the GC call in the Java CLI version, and also the possible gains from its parallelization. This will be tackled in the next chapter.

Chapter 4

Enhanced Java Version

This chapter revisits the Java CLI implementation of the DimReduction method, investigating the impact of the same enhancements applied in the Python-based implementation: removal of explicit Garbage Collection calls and parallelization based on a multithreaded approach. A performance comparison is conducted against the optimized Python code, leading to the recognition of the new, enhanced Java version as the most performant. The enhanced Java CLI version of the DimReduction method is then compared to alternative reference platforms from the literature of network inference methods.

4.1 Computational Environment and Datasets

All experiments documented in this chapter were performed in the **64-thread machine**. Some used the medium-size (400 genes) subset of the E. coli dataset (Network 3) from the DREAM5 Network Inference challenge, and others used the full dataset (4511 genes).

4.2 Java CLI without explicit GC

In the previous chapter it was found that by avoiding explicit calls to the Garbage Collector, the Python CLI version performance improved dramatically, and parallelization yielded the expected benefits. The next logical step is to apply the same optimizations

to the Java CLI version and study their effect. This is fundamental to ensure a fair comparison between the Python and Java CLI versions. For this comparison, the Java CLI version with the GC explicit call is named **Java-Serial-explicit-GC** and the version with the explicit GC call removed is designated as **Java-Serial-no-explicit-GC**.

Table 4.1: Performance Impact of Garbage Collection on Java CLI with 400 genes

Java CLI version	**Duration (hh:mm:ss)**	**Speedup**
Java-Serial-explicit-GC	0:38:00 (a)	—
Java-Serial-no-explicit-GC	0:00:20	114x

(a) from subsection 3.5.4

Table 4.1 shows the impact of the presence/absence of explicit GC calls with 400 genes. The impact of avoiding the GC calls is dramatic, with a speedup of 114x. This observation led to the decision of repeating the experiment with the full-size dataset, corresponding to a realistic usage scenario. The results are shown in Table 4.2.

Table 4.2: Performance Impact of Garbage Collection on Java CLI with 4511 genes

Java CLI version	**Duration (hh:mm:ss)**	**Speedup**
Java-Serial-explicit-GC	more than 2 days	—
Java-Serial-no-explicit-GC	00:42:26	$\gtrapprox$ 67x

With the full-dataset, the speedup generated by the removal of the explicit GC call is not as dramatic but is still very expressive. Thus, the conclusion is the same already taken with the Python CLI variant: explicit GC calls should be avoided (or at least minimized).

4.2.1 Comparison with the Python CLI version

At this stage, it becomes possible to compare the Java CLI version without the explicit call to GC, with the previous Python CLI versions (Serial and Parallel) – see Table 4.3.

Two main conclusions can be derived from Table 4.3: i) the Serial Java CLI variant keeps a speedup of 76-77x over the Serial Python CLI version, regardless of the data-set size; b) Even with the maximum number of threads supported by the tested system (64

Table 4.3: Python Java CLI vs Java CLI, without explicit GC

CLI version (dataset)	Duration (hh:mm:ss)	Speedup
Python-Serial-no-explicit-GC (400 genes)	00:25:31	—
Java-Serial-no-explicit-GC (400 genes)	00:00:20 (a)	76.55x
Python-Serial-no-explicit-GC (4511 genes)	54:38:17	—
Java-Serial-no-explicit-GC (4511 genes)	00:42:26 (b)	77.25x
Python-64Threads-no-explicit-GC (4511 genes)	01:38:31	—
Java-Serial-no-explicit-GC (4511 genes)	00:42:26 (b)	2.32x

(a) from Table 4.1; (b) from Table 4.2

threads), the parallel Python CLI variant is unable to be faster than the serial Java CLI; in fact, the latter is still being more than 2x faster than the former.

4.2.2 Alternative Python Interpreters

In a last effort to close the gap, performance-wise, between the Python CLI version and the Java CLI version, two final options were exploited:

- **PyPy:** PyPy is another implementation of Python (see `https://pypy.org/`) that has a Just-in-Time (JIT) compiler, which in many cases ensures a faster execution. Its major limitation, however, is that it cannot be used with the free-threaded mode (i.e. without the GIL), limiting the performance gains of a threaded Python CLI.

- **Python 3.14 (pre-release):** A Python version that theoretically provides the clearest route to efficient, parallel Python, as officially implements PEP 703 (see `https://peps.python.org/pep-0703/`), allowing the GIL to be optional in CPython.

Table 4.4 and Table 4.5 present the results of a comparison between the performance achieved by the Python interpreter used until now to run the Python CLI DimReduction (Python 3.13 without the GIL), and the two alternative interpreters introduced above (PyPy with the GIL, and Python 3.14 without the GIL). The tests were done with the full dataset (4511 genes), focusing 1st on the serial version of the Python CLI (Table 4.4) and 2nd on its parallel version (Sliced Groups approach) running with 64 threads (Table 4.4).

Table 4.4: Python Serial Execution Times.

Python-Serial-no-explicit-GC	**Duration (hh:mm:ss)**
PyPy (with GIL)	03:06:30
Python 3.14 (without GIL)	22:36:13
Python 3.13 (without GIL)	54:38:17 (a)

(a) from Table 4.3

Table 4.5: Python Parallel Execution Times.

Python-64Threads-no-explicit-GC	**Duration (hh:mm:ss)**
PyPy (with GIL)	04:48:14
Python 3.14 (without GIL)	03:52:14
Python 3.13 (without GIL)	01:38:31 (a)

(a) from Table 4.3

As can be seen in Table 4.4, the serial Python CLI version benefits substantially from the use of the PyPy interpreter. However, its running time (03:06:30) is still much larger than the comparable running time of the Java CLI version (00:42:23, from Table 4.3).

With regard to the parallel Python CLI, Table 4.5 proves that the Python interpreter that ensures faster execution is the one used so far (Python 3.13 without the GIL), thus putting to rest the issue of using an alternative interpreter to close the gap with the Java CLI, once, as it was seen in Table 4.3, even the serial Java CLI is faster than the fastest parallel Python CLI configuration (the one with the maximum of 64 threads).

4.3 Parallelization of the Java CLI version

After the previous tests, that confirmed the performance gap between the Java and the Python CLI code, the next move was to parallelize the Java code to maximize its performance potential. To this end, the same 3 approaches used for the Python code, to parallelize the loop that represents the main DimReduction hotspot, were also adopted to parallelize the Java code: *Slice Groups*, *Stride Groups* and *Dynamic Assignment*.

A performance evaluation of the new parallel Java CLI variant was then conducted, in the 64-thread system, using the full-size dataset (4511 genes).

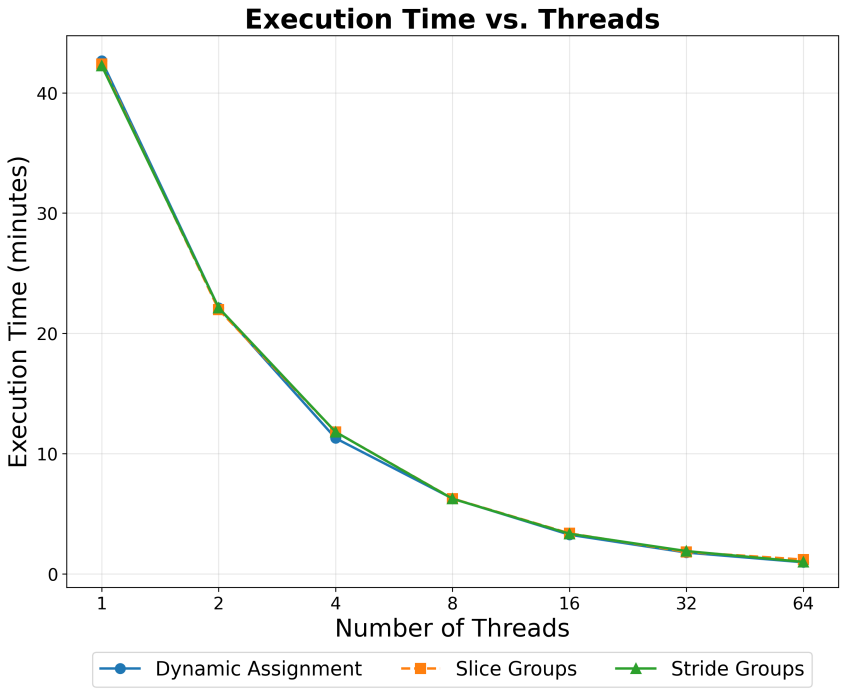


Figure 4.1: Real Execution Time vs. Number of Threads for the parallel Java CLI.

Figure 4.1 shows the execution time as a function of the number of threads, for the 3 parallelization approaches. The performance gains are similar for the three approaches, with the serial time (1 thread) of ≈43 minutes being reduced to just a mere ≈1 minute when using 64 threads. The *Dynamic Assignment* strategy shows a slight advantage in some situations, but the differences between all 3 strategies end up being mostly negligible.

To fairly evaluate the significance of the parallel times, it is necessary to compare the speedups achieved by the parallelized Java code against the theoretical speedups projected by Amdahl's Law. This requires determining the relative weight, in terms of execution time, of the parallelizable hotspot loop, when running the serial Java code. This measurement was done by instrumenting the serial code in order to sample the current time just before and right after the execution of that loop, and then dividing the time elapsed in between by the overall execution time of the program. This procedure returned a factor $p = 0.999$ (99.9%) for the parallelizable portion of the program, a very high and unusual value. This value (or a similar one) could have also been obtained by profiling, using the *perf* tool, but the direct instrumentation approach was chosen to avoid the

penalty of a longer execution time when running the application through *perf*.

Table 4.6 shows the maximum theoretical speedup, projected by Amdahl's Law, when $p = 0.999$, as well as the corresponding theoretical efficiency ($E_T(N) = S_T(N)/N$).

Table 4.6: Maximum theoretical speedup and efficient of the parallel Java CLI

Number of Threads (N)	**Theoretical Speedup** ($S_T(N)$)	**Theoretical Efficiency** ($E_T(N)$)
2	1.998	99.9 %
4	3.988	99.7 %
8	7.944	99.3 %
16	15.764	98.5 %
32	31.038	97.0 %
64	60.207	94.1 %

Figure 4.2 shows the plot of the real (effective) speedups reached by each of the 3 parallel approaches, based on the execution times of Figure 4.1. Figure 4.2 also shows the theoretical maximum speedup projected by Amdahl's Law, allowing a visual perception of the closeness of the real speedups to the theoretical maximum. This closeness is depicted in the 4th column of Table 4.7 for the *Dynamic Assignment* variant, the fastest of the 3 parallelization approaches implemented and tested.

Table 4.7: Dynamic Assignment Speedup and Efficiency

Number of Threads (N)	**Real Speedup** ($S_R(N)$)	**Real Efficiency** ($E_R(N)$)	**Efficiency Ratio** ($E_R(N)/E_T(N)$)
2	1.928	96.4%	96.5%
4	3.784	94.6%	94.9%
8	6.818	85.2%	85.8%
16	13.128	82.1%	83.4%
32	24.151	75.5%	77.8%
64	44.522	69.6%	74.0%

More specifically, and only for the *Dynamic Assignment*, Table 4.7 shows: i) the real speedup, $S_R(N) = T_R(1)/T_R(N)$, where $T_R(1)$ is the serial time and $T_R(N)$ is the time with N threads (already shown in Figure 4.1); ii) the real efficiency, $E_R(N) = S_R(N)/N$; iii) an Efficiency Ratio, $E_R(N)/E_T(N)$, that measures the closeness of the real efficiency

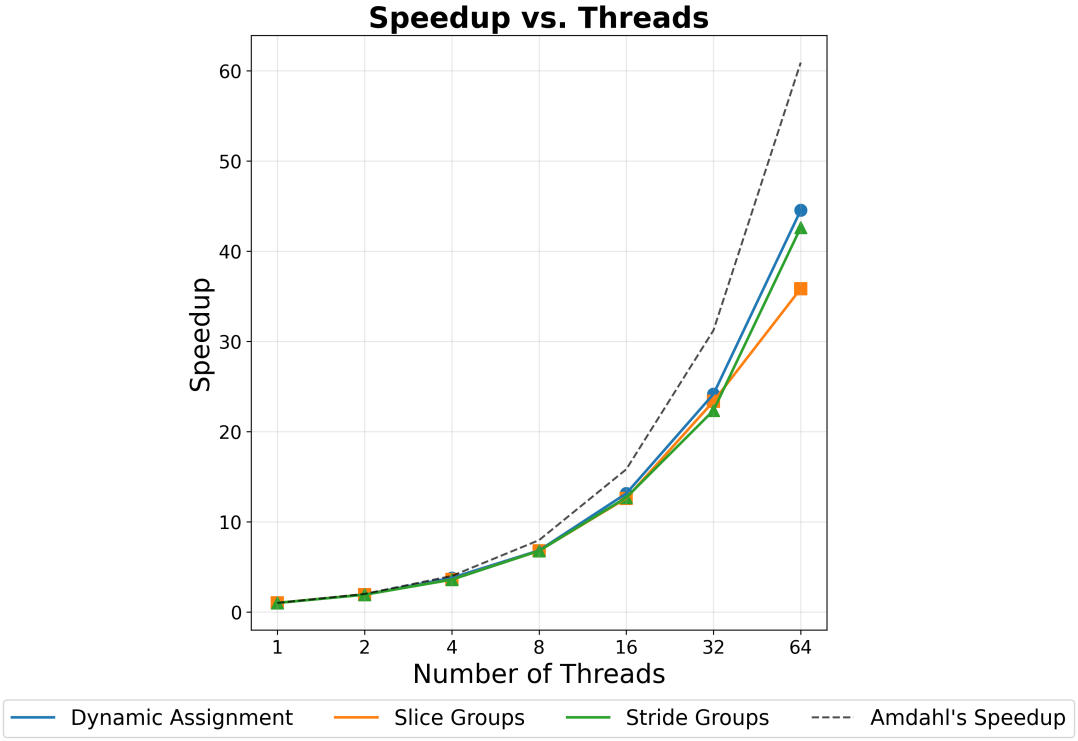


Figure 4.2: Speedups vs. Number of Threads for the parallel Java CLI.

to the maximum theoretical efficiency, a measure of the quality of the parallel code.

As can be seen, with 64 threads the parallel implementation is capable of running ≈ 45 times faster than the serial version, with a corresponding efficiency ratio of 74%. For a smaller number of threads, the efficiency ratio increases, showing that the parallel Java CLI application can take good advantage of CPUs with a low to moderate number of cores (those typically found in modern personal computers and workstations). Still, for server/HPC-class systems, the accelerations provided, although not maximized, are very welcomed, allowing to execute the application with large datasets in just a few minutes.

4.4 Comparison with Alternative Methods

In this section, Dim Reduction is compared with methods from the literature to better understand the positioning of this solution using SFS+MCE. Based on Zhao, He, Tang, *et al.* [24] and Marbach, Costello, Küffner, *et al.* [7], some methods were selected for this

comparison. These methods were categorized, by Zhao, He, Tang, *et al.* [24], between algorithms better for small real networks, called *Model-based*, for steady-state data problems, called *Information theory-based*, and to large-scale networks, the *Machine learning-based*.

4.4.1 Choice of Methods

Three methods were chosen to start with, to be applied to the dataset that has being used - the E. Coli dataset of Dream5. This is a large-scale and steady-state, which means *Information theory-based* and *Machine learning-based* methods would be prioritized. For the 1st category, **ARACNe** and **CLR** were previously compared to DimReduction SFFS + MCE in Lopes, Martins, and Cesar [25]. For the methods that convert the problems into feature selection like DimReduction, **GENIE3** is a method that considers each target gene, according to Huynh-Thu, Irrthum, Wehenkel, *et al.* [10], and was one of the best results to AUPR on E. Coli database by Marbach, Costello, Küffner, *et al.* [7].

Concerning these three methods, besides being individually available, the project GENECI from Segura-Ortiz, García-Nieto, Aldana-Montes, *et al.* [8] was found to make use of them, and other methods, to infer networks like Dream4 and Dream5 easier. To use the GENECI project, the documentation usage was used to make some tests to ensure input formats and solve environment dependencies. The main feature was the individual technique inference command, which was performed on the Dream4 first, since it was better documented and more evident in the Segura-Ortiz, García-Nieto, Aldana-Montes, *et al.* [8], and it would take less time because of the size of the dataset. These tests were essential to understand the format difference between both projects, and the compatibility or not of each method with multithreading. Only **KBOOST**, **TIGRESS** and **GENIE3** with Random Forest and Extra Tree were able to use more than one thread.

4.4.2 Tests Setup

The final output of DimReduction is a binary adjacency matrix, where the edges found are represented by the value 1. These edges are any connections with entropy lower

than the 'THRESHOLD' stipulated in the '.env' file. Thus, a new environment variable, called 'SAVE_FINAL_WEIGHT_DATA', was added to export the result as a weighted adjacency matrix, where the edges found are represented by the value $1 - entropy$ and the 'THRESHOLD' was set to 1, since it was intended the evaluation script to be able to set the threshold it needs to generate the metrics.

Although the Network 3 from Dream5 was already being used, to make the process more similar between both solutions some changes were made. First, on this challenge the transcription factors (TF) were provided to be used in the analysis, and the DimReduction code was changed to accept the TF list and ignore predictors that are not TF. Second, the input used by the GENECI project was used to DimReduction too, but changing between comma-separated file to tab-separated. Because of this new input, some changes in the environment configuration file were required, since the dataset was already transposed and has a header. After all these changes, the environment file became as in Listing C.2. An example of the configuration file used for the comparison experiments is in Listing C.2.

More format changes were required and made by external scripts. For DimReduction, the output needed to be converted as a list ordered by the weight of the edge, as in GENECI, and both changed to follow the tab-separated list of Matlab evaluating script offered by the Dream5 challenge (see `https://www.synapse.org/Synapse:syn2787214`).

4.4.3 Tests Execution

The real execution with Dream5 was prepared at this point and was done in the 64-threads machine. All methods were first executed with one thread, and then the ones supporting multithreading were re-executed using 64 threads. A cooldown pause of 15 minutes between the methods was enforced. However, it is important to note that TIGRESS was taking more than 5 days to resolve the inference with 64 threads, while the other methods did not even required a day. Therefore, this one was interrupted and discarded from the analysis of 1 and 64 threads, since it was not part of the three methods initially identified and had also been discarded in Segura-Ortiz, García-Nieto, Aldana-Montes, *et al.* [8],

when they were testing this dataset.

After the execution finished, some adjustments at the evaluation script were required, because some of the outputs were too big. The script needed to load the data in chunks of a quarter of the size of the input files. Some other changes were to generate some extra graphs to performance metrics across different confidence thresholds.

4.4.4 Quality Analysis

Confusion Matrices

The quality of the predictions is measured by starting with the confusion matrices of each of the methods at various levels of confidence. The methodology gives a clear picture of trade-offs of network inference. An example is a technique that tries to make conclusions about all genes, which would get a perfect recall (100% True Positives) but with a very large number of False Positives (FP), which would result in very low precision.

Figure 4.3 to Figure 4.5 are depictions of the heatmaps of the confusion matrices at the confidence levels of 75% (Figure 4.3), 50% (Figure 4.4, and 25% (Figure 4.5), respectively. The low TP values here are one of indicatives about the difficulty of the challenge. This dataset has 2066 true positive edges in the gold standard.

These depictions indicate that there are two different profiles in the approaches considering the confidence level. The majority of procedures (e.g., ARACNE, CLR, C3NET) are conservative, find very small numbers of True Positives (TP), yet produce virtually no False Positives (FP), except for C3NET, that stayed stable during all threshold, with a competitive values of both metrics.

On the other hand, DimReduction and PCIT are more sensitive and they will capture more TPs. But such heightened sensitivity is achieved at a very high price, namely, a considerable increase in FP, which can affect the precision of the results. Indicatively, as the confidence threshold of DimReduction is lowered from 75% to 25%, the number of TPs increases from 27 to 91, and the FPs more than double from 512 to 1185. PCIT has a parallel behavior that is more extreme.

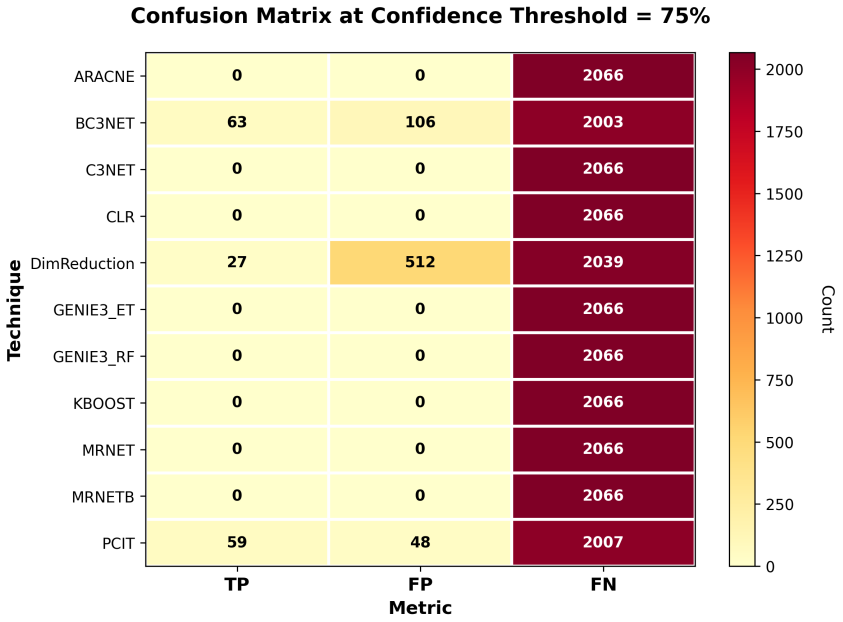


Figure 4.3: Confusion Matrix Heatmap at 75% Confidence Threshold.

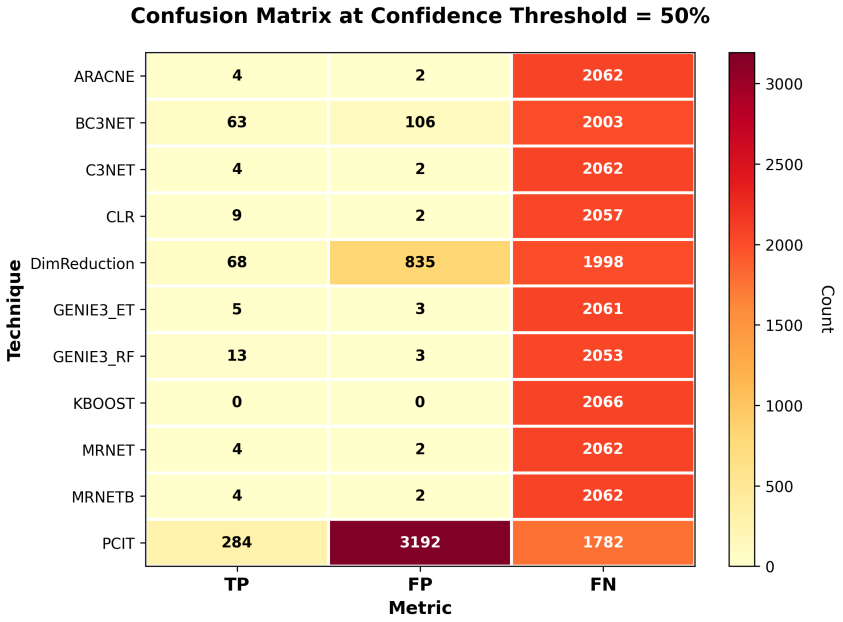


Figure 4.4: Confusion Matrix Heatmap at 50% Confidence Threshold.

An important point to note throughout all procedures is that the number of False Negatives (FN) is always high and it stands at $\approx$ 2000. This shows that much of the actual regulatory interaction is not being identified using any of the methods, contributing

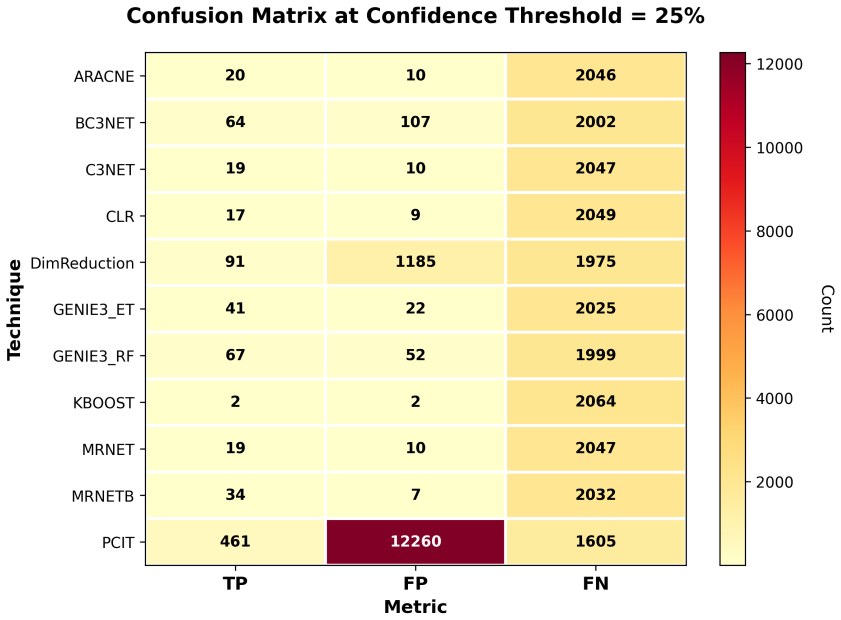


Figure 4.5: Confusion Matrix Heatmap at 25% Confidence Threshold.

to the complexity of the dataset and the difficulty of the inference process. This implies a more in-depth discussion of balanced metrics, such as the F1 score (see next).

F1 Score Curve

The F1 score, which represents the harmonic mean of *precision* and *recall*, demonstrates the model's balanced performance across different confidence thresholds, since *recall* is the true positive rate. The generally low F1 scores across all methods, values less than 0.15, as shown in Figure 4.6, further underscore the difficulty of the challenge. While PCIT showed a high number of TPs in the heatmaps, its F1 score drops quickly as confidence decreases below 55%, confirming that its high *recall* comes at the cost of low precision. DimReduction maintains a more stable, albeit lower, F1 score, consistent with its behavior of having a constantly high number of false positives.

Precision-Recall Curve

Analyzing the Precision-Recall curve of the performance on positive prediction, the Dim-Reduction with SFS+MCE had a low performance compared to the other methods, as

shown in Figure 4.7. But the whole scenario shows how difficulty it was to predict correctly, since all methods were only precise with low true predictions.

ROC Curve

Analyzing the ROC curve of the performance on learn to classify, the DimReduction with SFS+MCE had a low performance compared to other methods, as shown in Figure 4.7. But the whole scenario still shows how difficulty it was to distinguish true positive from false positive, since most methods were close to a Random Classifier.

AUROC and AUPR

To better understand this performance scenario, the **AUROC** (*Area Under the ROC*) and **AUPR** (*Area Under the Precision Recall*) were represented graphically, as shown in Figure 4.9. The visualization shows the complexity of this dataset again, since most of them achieved only moderate performance levels.

4.4.5 Execution Time Analysis

In addition to quality measures, the execution time of each method was monitored on the complete DREAM5 Network 3 dataset (4511 genes), allowing an analysis of computational performance, as shown in Figure 4.10.

When considering execution with only one thread on a 64-core machine, DimReduction ranked fourth, significantly ahead of slowest methods, such as GENIE3. In the same graph, it is possible to see that the use of 64 threads makes DimReduction more interesting with almost no waiting time, demonstrating some scalability with multiple threads.

4.4.6 Balancing Quality and Execution Time

The choice of a technique can be influenced by both the quality of the results and the speed of execution. Thus, a weighted score was defined to rank the techniques. The goal is to allow the user (scientist) to balance speed and quality (the metric of interest).

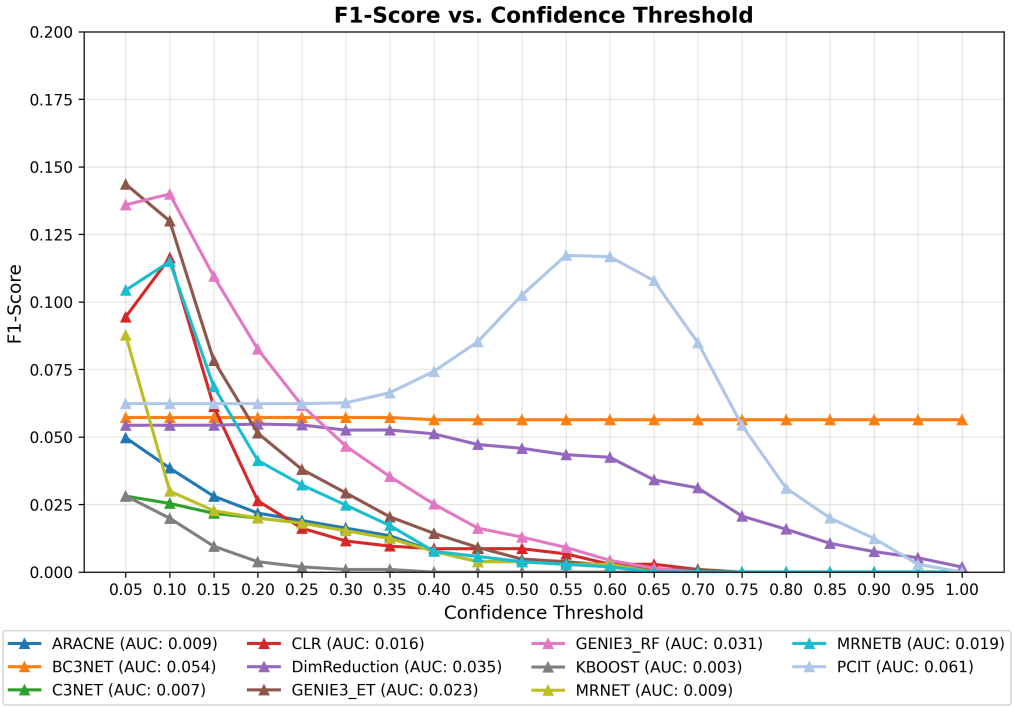


Figure 4.6: F1 Score vs. Confidence Threshold.

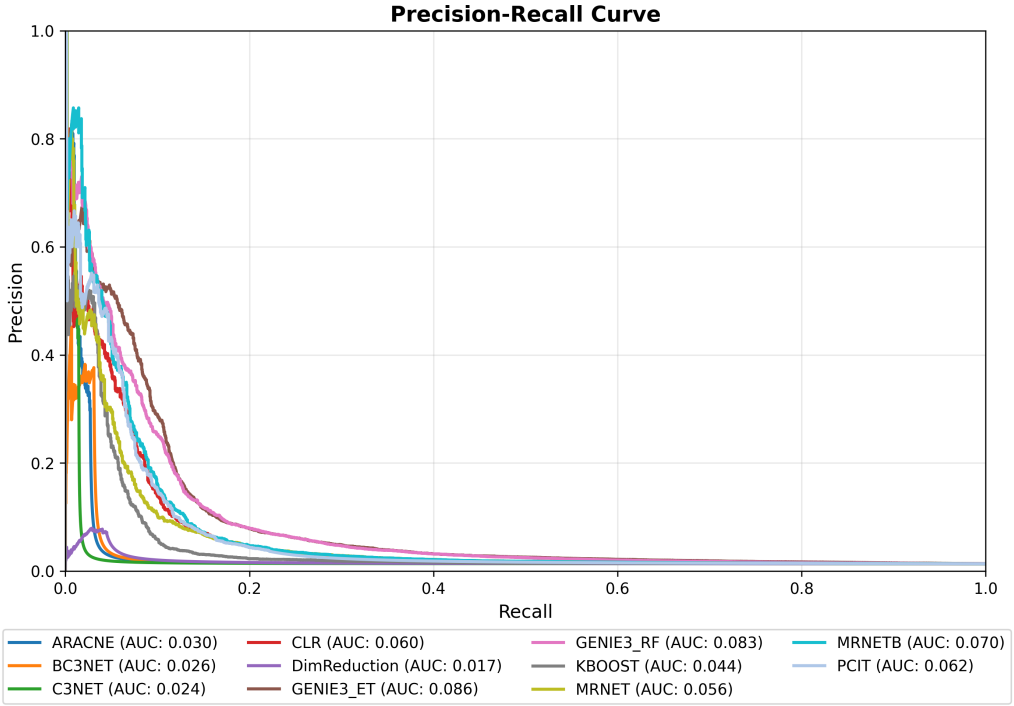


Figure 4.7: Precision-Recall Curve.

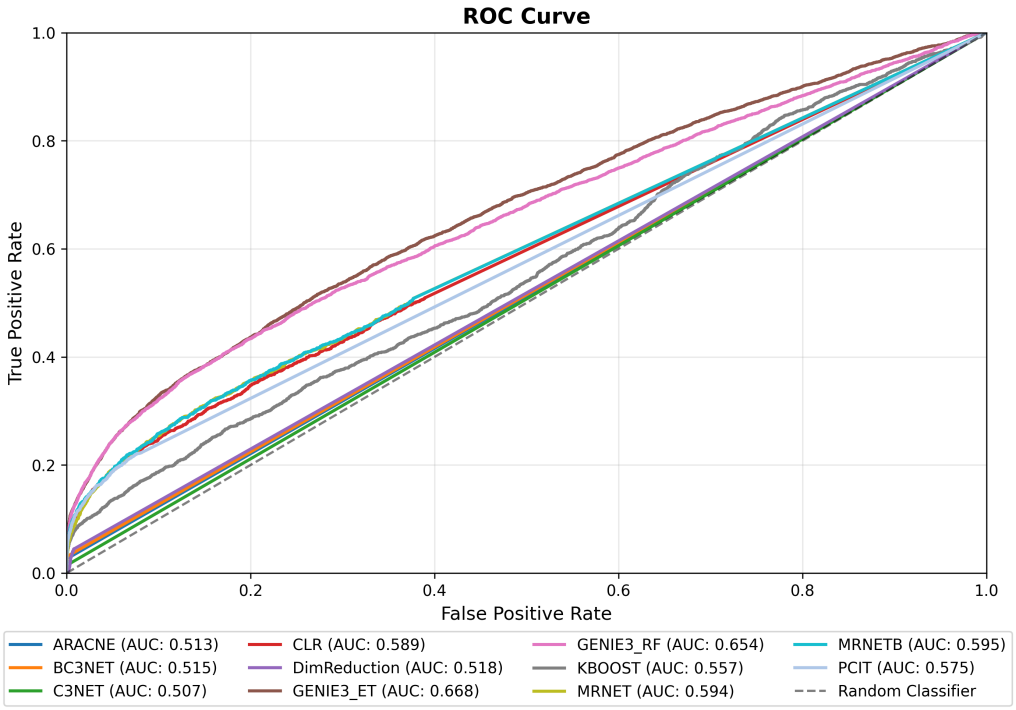


Figure 4.8: ROC Curve.

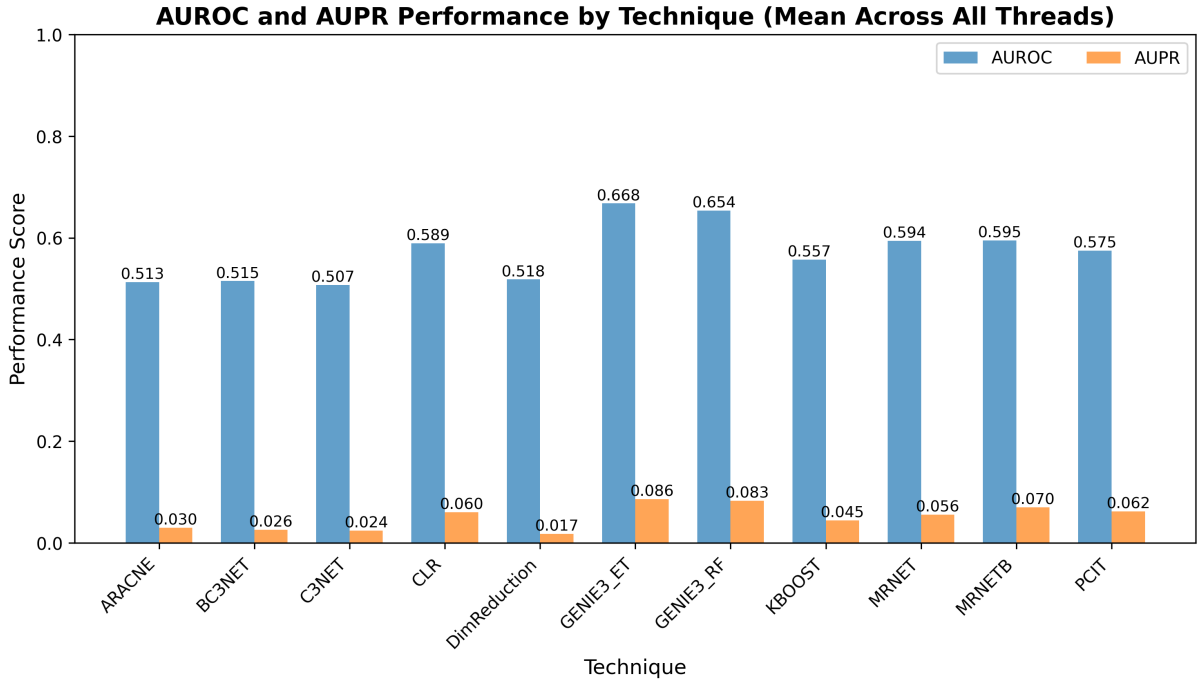


Figure 4.9: AUROC and AUPR Scores for All Methods.

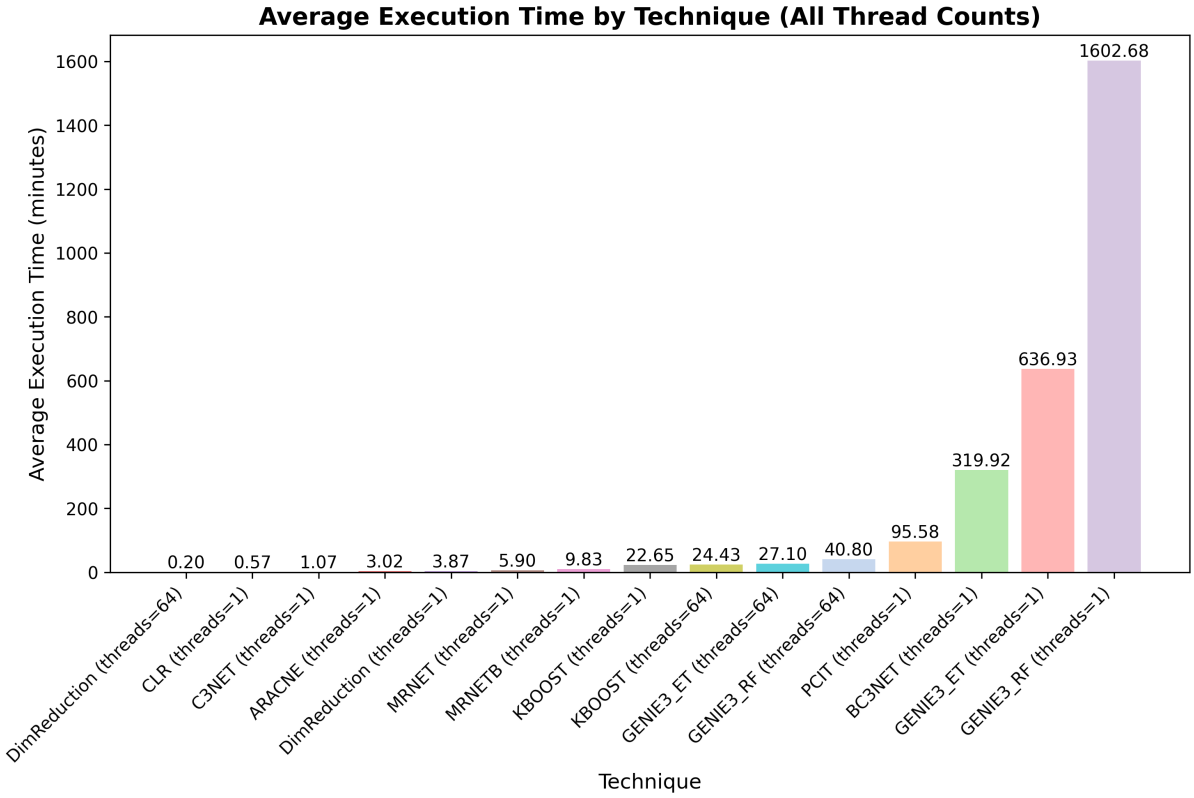


Figure 4.10: Execution Time Comparison for All Methods.

As before, all results in this section correspond to executions on the complete DREAM5 Network 3 dataset (4511 genes) using the 64-thread machine. To ensure a fair comparison with the metrics (which range from 0 to 1), the execution time was normalized to a range of 0 to 1. The ranking formula, which combines the metric score (Metric) and the normalized Relative Speed, is presented in Equation 4.1 and Equation 4.2.

$$\text{Score}_{\text{Weighted}} = (\alpha \times \text{Metric}) + (1 - \alpha) \times \text{Relative Speed} \tag{4.1}$$

$$\text{Relative Speed} = \frac{\text{Time}_{\text{Worst}} - \text{Time}_{\text{Method}}}{\text{Time}_{\text{Worst}}} \tag{4.2}$$

In Equation 4.1, α is the weight assigned to quality, ranging from 0 to 1. In Equation 4.2, $\text{Time}_{\text{Worst}}$ is the longest execution time among all methods.

The results of this new evaluation metric, which represents the trade-off between

quality metric and speed, shows that for most methods, in the trade-off involving AUPR, the scores remained very close, both in terms of time and quality, except for three methods that presented significant discrepancies in time, as shown in Figure 4.11.

Similarly, the trade-off involving AUROC, shown in Figure 4.12, demonstrates a comparable pattern but with slightly greater dispersion in quality metrics, indicating more varied performance optimization points across different methods.

With a breakdown of the scores obtained for AUPR weights of 25%, 50% and 75%, as shown in Table 4.8, reveals that DimReduction with one thread remains in the last positions for most weight values. The DimReduction with 64 threads starts with a moderate ranking (due to time prioritization), but its position decrease as the weight on quality increases. Similar behavior is observed for AUROC, as shown in Table 4.9.

To consolidate the performance in the trade-off, the area under the curve (AUC) of the weighted efficiency graphs (AUWP) was calculated, resulting in a final score for each technique, shown in Figure 4.13. GENIE3 with Extra Tree stands out as the best method in this new score, while GENIE3 with Random Forest is below average. DimReduction (with both 1 and 64 threads) presents an average performance in this metric, varying between the 10th and 11th position in the final AUWP ranking to AUPR, between the 8th and 9th position in the final AUWP ranking to AUROC, as shown in Table 4.10.

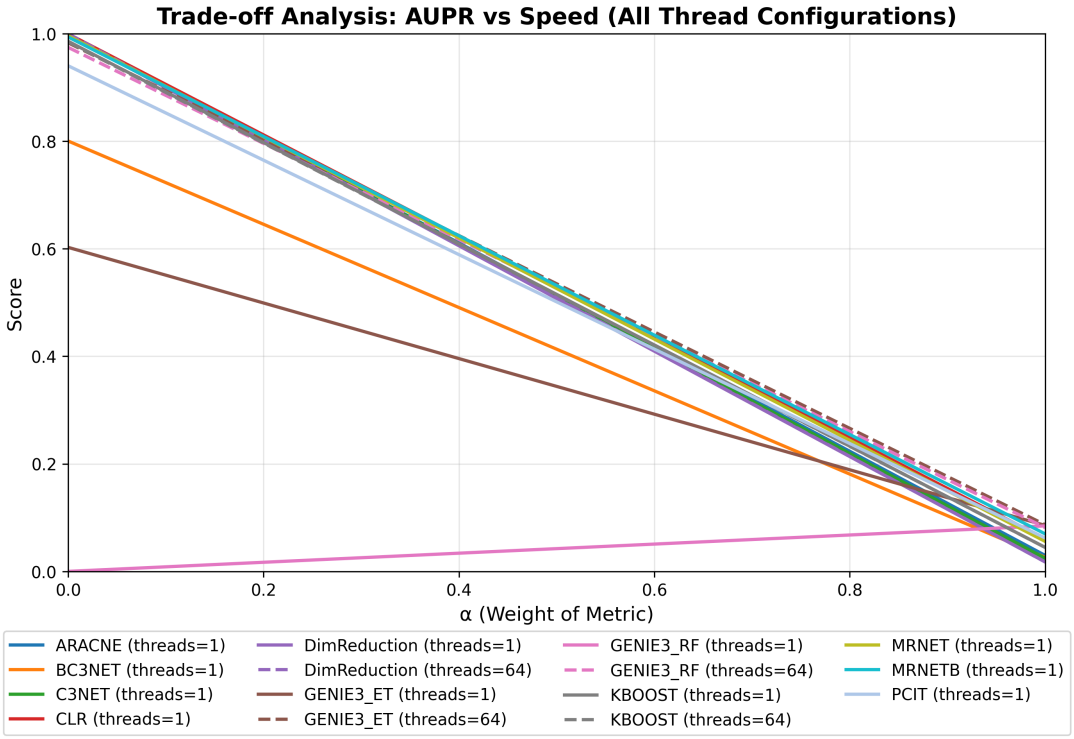


Figure 4.11: Trade-off Analysis Between AUPR and Execution Time.

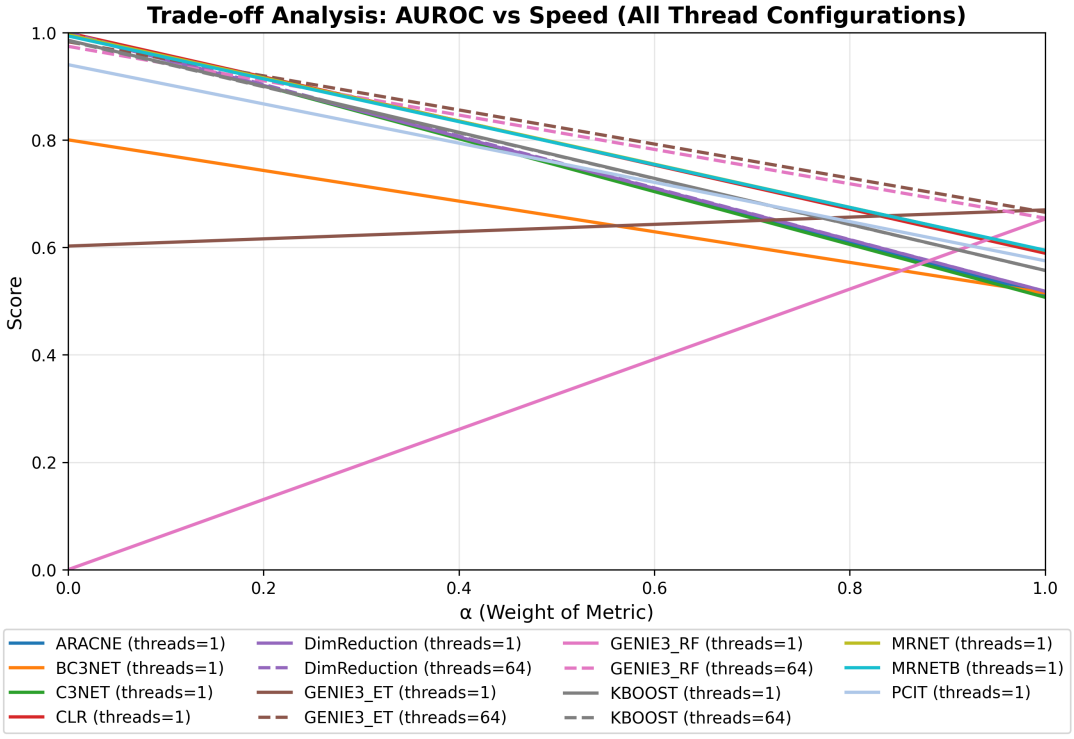


Figure 4.12: Trade-off Analysis Between AUROC and Execution Time.

Table 4.8: AUPR Trade-off Rankings.

25% weight	**50% weight**	**75% weight**
CLR (1) - 0.765	GENIE3_ET (64) - 0.535	GENIE3_ET (64) - 0.311
MRNETB (1) - 0.763	MRNETB (1) - 0.532	GENIE3_RF (64) - 0.305
MRNET (1) - 0.761	CLR (1) - 0.530	MRNETB (1) - 0.301
GENIE3_ET (64) - 0.759	GENIE3_RF (64) - 0.528	CLR (1) - 0.295
ARACNE (1) - 0.756	MRNET (1) - 0.526	MRNET (1) - 0.291
C3NET (1) - 0.756	KBOOST (1) - 0.515	PCIT (1) - 0.282
DimReduction (64) - 0.754	KBOOST (64) - 0.515	KBOOST (1) - 0.280
DimReduction (1) - 0.753	ARACNE (1) - 0.514	KBOOST (64) - 0.280
GENIE3_RF (64) - 0.751	C3NET (1) - 0.512	ARACNE (1) - 0.272
KBOOST (1) - 0.751	**DimReduction (64)** - 0.509	C3NET (1) - 0.268
KBOOST (64) - 0.750	**DimReduction (1)** - 0.508	**DimReduction (64)** - 0.263
PCIT (1) - 0.721	PCIT (1) - 0.501	**DimReduction (1)** - 0.262
BC3NET (1) - 0.607	BC3NET (1) - 0.413	BC3NET (1) - 0.220
GENIE3_ET (1) - 0.473	GENIE3_ET (1) - 0.344	GENIE3_ET (1) - 0.215
GENIE3_RF (1) - 0.021	GENIE3_RF (1) - 0.042	GENIE3_RF (1) - 0.063

Table 4.9: AUROC Trade-off Rankings.

25% weight	**50% weight**	**75% weight**
GENIE3_ET (64) - 0.904	GENIE3_ET (64) - 0.824	GENIE3_ET (64) - 0.745
CLR (1) - 0.897	GENIE3_RF (64) - 0.815	GENIE3_RF (64) - 0.735
MRNET (1) - 0.896	MRNET (1) - 0.795	MRNET (1) - 0.695
GENIE3_RF (64) - 0.895	CLR (1) - 0.794	MRNETB (1) - 0.695
MRNETB (1) - 0.894	MRNETB (1) - 0.794	CLR (1) - 0.692
DimReduction (64) - 0.879	KBOOST (1) - 0.772	PCIT (1) - 0.666
KBOOST (1) - 0.879	KBOOST (64) - 0.771	KBOOST (1) - 0.664
KBOOST (64) - 0.878	**DimReduction (64)** - 0.759	KBOOST (64) - 0.664
DimReduction (1) - 0.878	**DimReduction (1)** - 0.758	GENIE3_ET (1) - 0.653
ARACNE (1) - 0.877	PCIT (1) - 0.758	**DimReduction (64)** - 0.639
C3NET (1) - 0.876	ARACNE (1) - 0.756	**DimReduction (1)** - 0.638
PCIT (1) - 0.849	C3NET (1) - 0.753	ARACNE (1) - 0.634
BC3NET (1) - 0.729	BC3NET (1) - 0.658	C3NET (1) - 0.630
GENIE3_ET (1) - 0.619	GENIE3_ET (1) - 0.636	BC3NET (1) - 0.586
GENIE3_RF (1) - 0.163	GENIE3_RF (1) - 0.327	GENIE3_RF (1) - 0.490

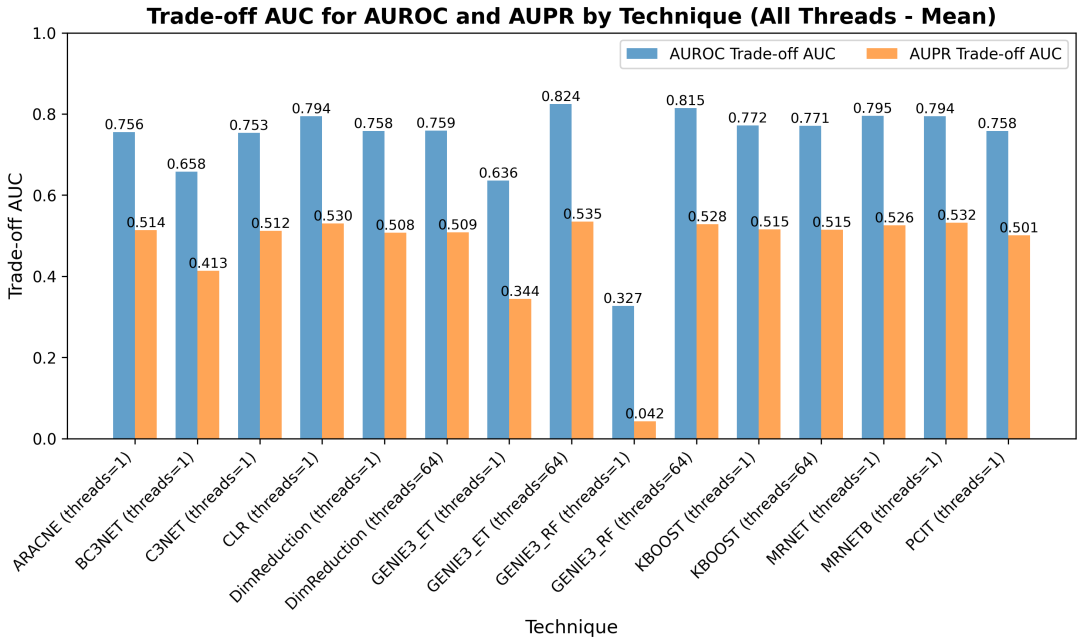


Figure 4.13: Consolidated Trade-off AUC Scores (AUWP) for All Methods.

Table 4.10: Consolidated AUWP Rankings.

Rank	AUWP-AUPR	AUWP-AUROC
1	0.535 GENIE3_ET (threads=64)	0.824 GENIE3_ET (threads=64)
2	0.532 MRNETB (threads=1)	0.815 GENIE3_RF (threads=64)
3	0.530 CLR (threads=1)	0.795 MRNET (threads=1)
4	0.528 GENIE3_RF (threads=64)	0.794 CLR (threads=1)
5	0.526 MRNET (threads=1)	0.794 MRNETB (threads=1)
6	0.515 KBOOST (threads=1)	0.772 KBOOST (threads=1)
7	0.515 KBOOST (threads=64)	0.771 KBOOST (threads=64)
8	0.514 ARACNE (threads=1)	0.759 **DimReduction (threads=64)**
9	0.512 C3NET (threads=1)	0.758 **DimReduction (threads=1)**
10	0.509 **DimReduction (threads=64)**	0.758 PCIT (threads=1)
11	0.508 **DimReduction (threads=1)**	0.756 ARACNE (threads=1)
12	0.501 PCIT (threads=1)	0.753 C3NET (threads=1)
13	0.413 BC3NET (threads=1)	0.658 BC3NET (threads=1)
14	0.344 GENIE3_ET (threads=1)	0.636 GENIE3_ET (threads=1)
15	0.042 GENIE3_RF (threads=1)	0.327 GENIE3_RF (threads=1)

Chapter 5

Conclusion

This work began with the goal of optimizing the DimReduction tool, starting with its conversion to a command-line interface and Python. To guide optimization efforts, performance analyses designed to identify key computational bottlenecks were carried out. Although the initial analysis correctly identified a specific function as the main hotspot, when attempting to implement parallelism in Python, an unexpected challenge emerged: adding multiple threads did not deliver the expected performance gains.

Further investigation revealed that the true source of the problem laid not within the function flow, but in explicit and frequent calls to the Garbage Collector. Removing this call significantly improved performance. But the most striking discovery came when applying the same optimization to the Java version: an execution time, previously estimated at more than 2 days for that version, was reduced to just 42 minutes. This surprising result established the optimized Java version as the best version, even after some other improvements were made to the Python version, redefining the project's direction to focus on the optimization and parallelization of the Java version.

The main contribution of this work, besides producing a faster tool, lies in demonstrating an optimization methodology where the use of performance analysis tools, such as *perf* and *Hotspot*, was essential to uncovering the true nature of the computational bottleneck. A important lesson here is that in Bioinformatics applications that manipulate large volumes of data, critical bottlenecks can hide in memory management operations,

easily being overlooked in superficial analyses.

To validate its relevance, the parallelized Java DimReduction tool was subjected to a rigorous comparison with other genetic network inference methods in the literature, using the DREAM5 test dataset. The results revealed a trade-off between speed and prediction quality. While methods like GENIE3 achieved superior performance in quality statistics, such as AUPR and AUROC, DimReduction stood out with a excellent computational performance, being one of the fastest tools. A thoughtful performance analysis, which balances execution time and quality, positioned DimReduction as a competitive solution, especially valuable in scenarios where analysis speed is a critical factor for researchers.

5.1 Future Work

Several research and development avenues are envisioned to expand the impact and functionality of the DimReduction tool:

1. It is essential to validate the operation of other configurations of the original tool in the new CLI interface with parallelism, such as the SFFS algorithm, rigorously testing its performance and scalability with multiple threads.

2. Direct integration of DimReduction into the GENECI project would increase its visibility and facilitate its adoption by the community.

3. The tool itself could be enriched by implementing other methods from the literature, solidifying it as an even more comprehensive platform for genetic network analysis.

4. The tool could also be enriched with support to other forms of parallel execution, namely message passing for a distributed parallel execution in HPC clusters, or accelerated execution in co-processors (namely GP-GPUs).

Finally, a scientific publication is also planned, covering the main aspects of this work.

Bibliography

[1] F. M. Lopes, D. C. Martins, and R. M. Cesar, "Feature selection environment for genomic applications," *BMC Bioinformatics*, vol. 9, no. 1, p. 451, Oct. 2008, ISSN: 1471-2105. DOI: 10.1186/1471-2105-9-451. [Online]. Available: https://doi.org/10.1186/1471-2105-9-451.

[2] F. M. Lopes and R. M. César Júnior, "Redes complexas de expressão gênica: Síntese, identificação, análise e aplicações," Portuguese, Ph.D. dissertation, Universidade de São Paulo, Brazil, South America, Jan. 2011. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=4b97c881-dee4-3aa0-bec7-af3f559e117c.

[3] G. Wilk and Z. Wlodarczyk, "Example of a possible interpretation of tsallis entropy," *Physica A: Statistical Mechanics and its Applications*, vol. 387, no. 19, pp. 4809–4813, 2008, ISSN: 0378-4371. DOI: https://doi.org/10.1016/j.physa.2008.04.022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378437108003920.

[4] P. Somol, P. Pudil, J. Novovičová, and P. Paclík, "Adaptive floating search methods in feature selection," *Pattern Recognition Letters*, vol. 20, no. 11, pp. 1157–1163, 1999, ISSN: 0167-8655. DOI: https://doi.org/10.1016/S0167-8655(99)00083-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167865599000835.

[5] V. Berisha, C. Krantsevich, P. R. Hahn, *et al.*, "Digital medicine and the curse of dimensionality," *npj Digital Medicine*, vol. 4, no. 1, p. 153, Oct. 2021, ISSN: 2398-6352. DOI: 10.1038/s41746-021-00521-5. [Online]. Available: https://doi.org/10.1038/s41746-021-00521-5.

[6] S. Guo, Q. Jiang, L. Chen, and D. Guo, "Gene regulatory network inference using pls-based methods," *BMC Bioinformatics*, vol. 17, no. 1, p. 545, Dec. 2016, ISSN: 1471-2105. DOI: 10.1186/s12859-016-1398-6. [Online]. Available: https://doi.org/10.1186/s12859-016-1398-6.

[7] D. Marbach, J. C. Costello, R. Küffner, *et al.*, "Wisdom of crowds for robust gene network inference," *Nature Methods*, vol. 9, no. 8, pp. 796–804, 2012, ISSN: 1548-7091. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=f518ca1a-3b81-35de-9a49-c70f578318fe.

[8] A. Segura-Ortiz, J. García-Nieto, J. F. Aldana-Montes, and I. Navas-Delgado, "Geneci: A novel evolutionary machine learning consensus-based approach for the inference of gene regulatory networks," *Computers in Biology and Medicine*, vol. 155, p. 106 653, 2023, ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.2023.106653. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S001048252300118X.

[9] G. Altay and F. Emmert-Streib, "Inferring the conservative causal core of gene regulatory networks," *BMC Systems Biology*, vol. 4, no. 1, p. 132, Sep. 2010, ISSN: 1752-0509. DOI: 10.1186/1752-0509-4-132. [Online]. Available: https://doi.org/10.1186/1752-0509-4-132.

[10] V. A. Huynh-Thu, A. Irrthum, L. Wehenkel, and P. Geurts, "Inferring regulatory networks from expression data using tree-based methods.," *PloS one*, vol. 5, no. 9, null–null, 2010, ISSN: 1932-6203; Electronic. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=ca905c56-5438-3f33-beab-aeb64a5c5524.

[11] L. F. Iglesias-Martinez, B. De Kegel, and W. Kolch, "Kboost: A new method to infer gene regulatory networks from gene expression data," *Scientific Reports*, vol. 11, no. 1, p. 15 461, Jul. 2021, ISSN: 2045-2322. DOI: 10.1038/s41598-021-94919-6. [Online]. Available: https://doi.org/10.1038/s41598-021-94919-6.

[12] P. E. Meyer, K. Kontos, F. Lafitte, and G. Bontempi, "Information-theoretic inference of large transcriptional regulatory networks," *EURASIP Journal on Bioinformatics and Systems Biology*, vol. 2007, pp. 1–9, 2006, ISSN: 1687-4145. DOI: 10.1155/2007/79879.

[13] A. A. Margolin, I. Nemenman, K. Basso, *et al.*, "Aracne: An algorithm for the reconstruction of gene regulatory networks in a mammalian cellular context," *BMC Bioinformatics*, vol. 7, no. 1, S7, Mar. 2006, ISSN: 1471-2105. DOI: 10.1186/1471-2105-7-S1-S7. [Online]. Available: https://doi.org/10.1186/1471-2105-7-S1-S7.

[14] J. J. Faith, B. Hayete, J. T. Thaden, *et al.*, "Large-scale mapping and validation of escherichia coli transcriptional regulation from a compendium of expression profiles," *PLOS Biology*, vol. 5, no. 1, pp. 1–13, Jan. 2007. DOI: 10.1371/journal.pbio.0050008. [Online]. Available: https://doi.org/10.1371/journal.pbio.0050008.

[15] R. de Matos Simoes and F. Emmert-Streib, "Bagging statistical network inference from large-scale gene expression data," *PLOS ONE*, vol. 7, no. 3, pp. 1–11, Mar. 2012. DOI: 10.1371/journal.pone.0033624. [Online]. Available: https://doi.org/10.1371/journal.pone.0033624.

[16] P. Meyer, D. Marbach, S. Roy, and M. Kellis, "Information-theoretic inference of gene networks using backward elimination.," English, in *BIOCOMP'10*, 700-705, CPS, 2010. HDL: https://orbi.uliege.be/2268/163899.

[17] A. Reverter and E. K. F. Chan, "Combining partial correlation and an information theory approach to the reversed engineering of gene co-expression networks," *Bioinformatics*, vol. 24, no. 21, pp. 2491–2497, Sep. 2008, ISSN: 1367-4803. DOI: 10.1093/

bioinformatics/btn482. eprint: https://academic.oup.com/bioinformatics/article-pdf/24/21/2491/49055106/bioinformatics_24_21_2491.pdf. [Online]. Available: https://doi.org/10.1093/bioinformatics/btn482.

[18] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using gpu architectures," *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014. DOI: 10.4208/cicp.110113.010813a.

[19] M. D. McCool, "Scalable programming models for massively multicore processors," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 816–831, 2008. DOI: 10.1109/JPROC.2008.917731.

[20] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, Mar. 2003, ISSN: 0360-0300. DOI: 10.1145/641865.641867. [Online]. Available: https://doi.org/10.1145/641865.641867.

[21] S. Tang, B.-S. Lee, and B. He, "Speedup for multi-level parallel computing," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, pp. 537–546. DOI: 10.1109/IPDPSW.2012.72.

[22] X.-H. Sun and L. Ni, "Another view on parallel speedup," in *Supercomputing '90:Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, 1990, pp. 324–333. DOI: 10.1109/SUPERC.1990.130037.

[23] X.-H. Sun and J. L. Gustafson, "Toward a better parallel performance metric," *Parallel Computing*, vol. 17, no. 10, pp. 1093–1109, 1991, Benchmarking of high performance supercomputers, ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(05)80028-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819105800286.

[24] M. Zhao, W. He, J. Tang, Q. Zou, and F. Guo, "A comprehensive overview and critical evaluation of gene regulatory network inference technologies.," *Briefings in*

Bioinformatics, vol. 22, no. 5, pp. 1–15, Sep. 2021, ISSN: 1467-5463. DOI: 10.1093/bib/bbab009. [Online]. Available: https://doi.org/10.1093/bib/bbab009.

[25] F. M. Lopes, D. C. Martins, and R. M. Cesar, "Comparative study of grns inference methods based on feature selection by mutual information," in *2009 IEEE International Workshop on Genomic Signal Processing and Statistics*, 2009, pp. 1–4. DOI: 10.1109/GENSIPS.2009.5174334.

Appendix A

DimReduction Quick Test-Drive

The DimReduction tool is provided as a set of JAR files, available at `https://sourceforge.net/projects/dimreduction/`.

The commands in Listing A.1 were used to compile and run the tool.

```
javac -d "out/production/java-dimreduction" \
      -cp "./lib/*:./out/production/java-dimreduction" \
      src/**/*.java;
cp -r "src/img" "out/production/java-dimreduction/";
java -cp "./lib/*:./out/production/java-dimreduction" fs.Main;
```

Listing A.1: Commands to compile and run the JAVA GUI-based DimReduction

The gene expression data from network 3 of the DREAM5 Network Inference Challenge was used to exercise the use of the software with real-world genomic datasets (see Section 3.2 for details). In this dataset, the genes are distributed in columns and samples in rows.

DimReduction was able to load the data and perform feature selection, but a filter was applied, so it doesn't take too much time. The test takes only the first 40 genes and the software was able to load the data and perform the network inference as shown below.

The first step was to fill the input data tab, as shown in Figure A.1, with:

- **Input Data:** The dataset was loaded from a file. The file selected was the one with the first 40 genes of the DREAM5 Network 3 dataset.

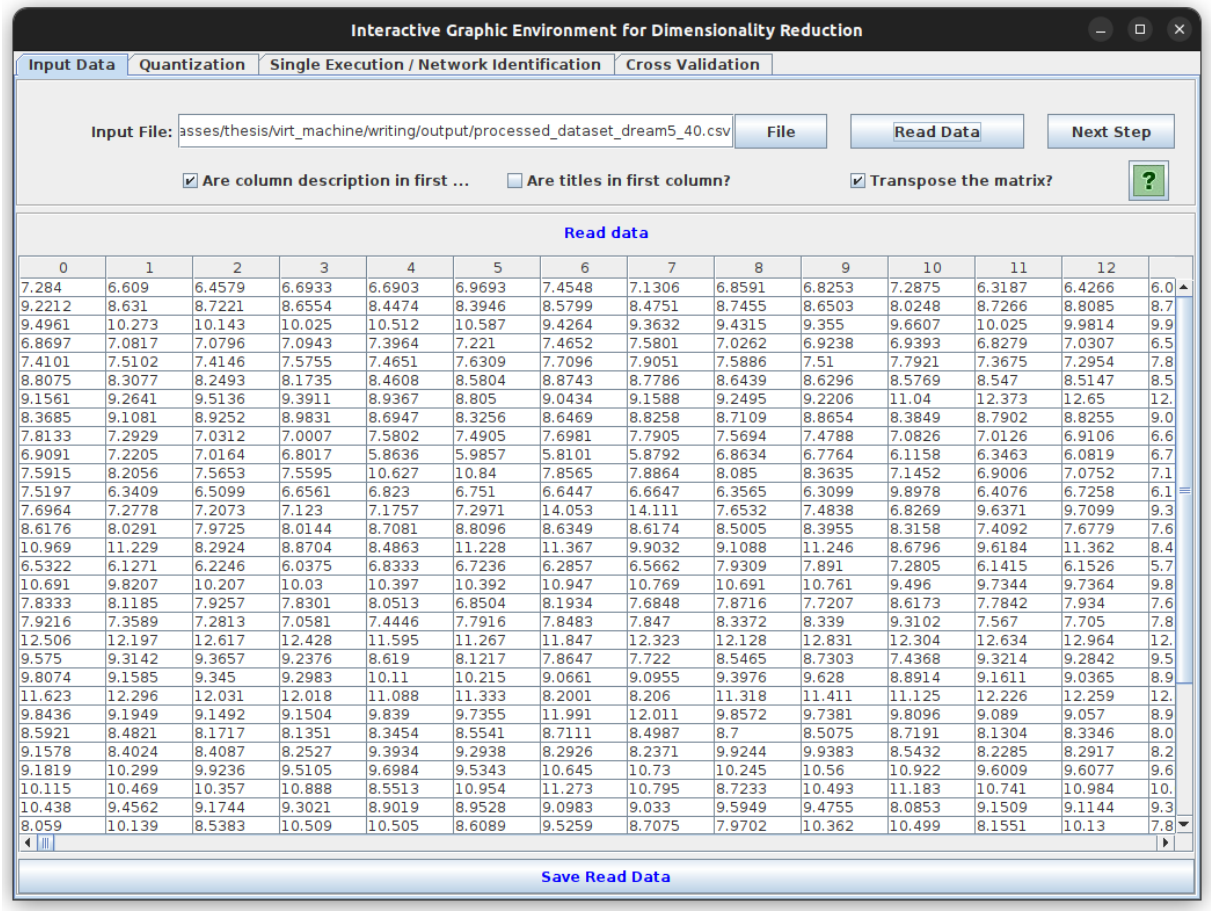


Figure A.1: DimReduction Input Data tab.

- **Are column description in first row?:** Yes. The first row of the dataset contains the names of the genes.

- **Are titles in first column?:** No. The first column of the dataset does not contain the names of the samples.

- **Transpose the matrix?:** Yes. The matrix was transposed so that the samples are in the columns and the genes in the rows.

The next step was to fill the Quantization tab, as shown in Figure A.2, with:

- **Quantity of Values:** 2. The data was quantized into 2 levels.

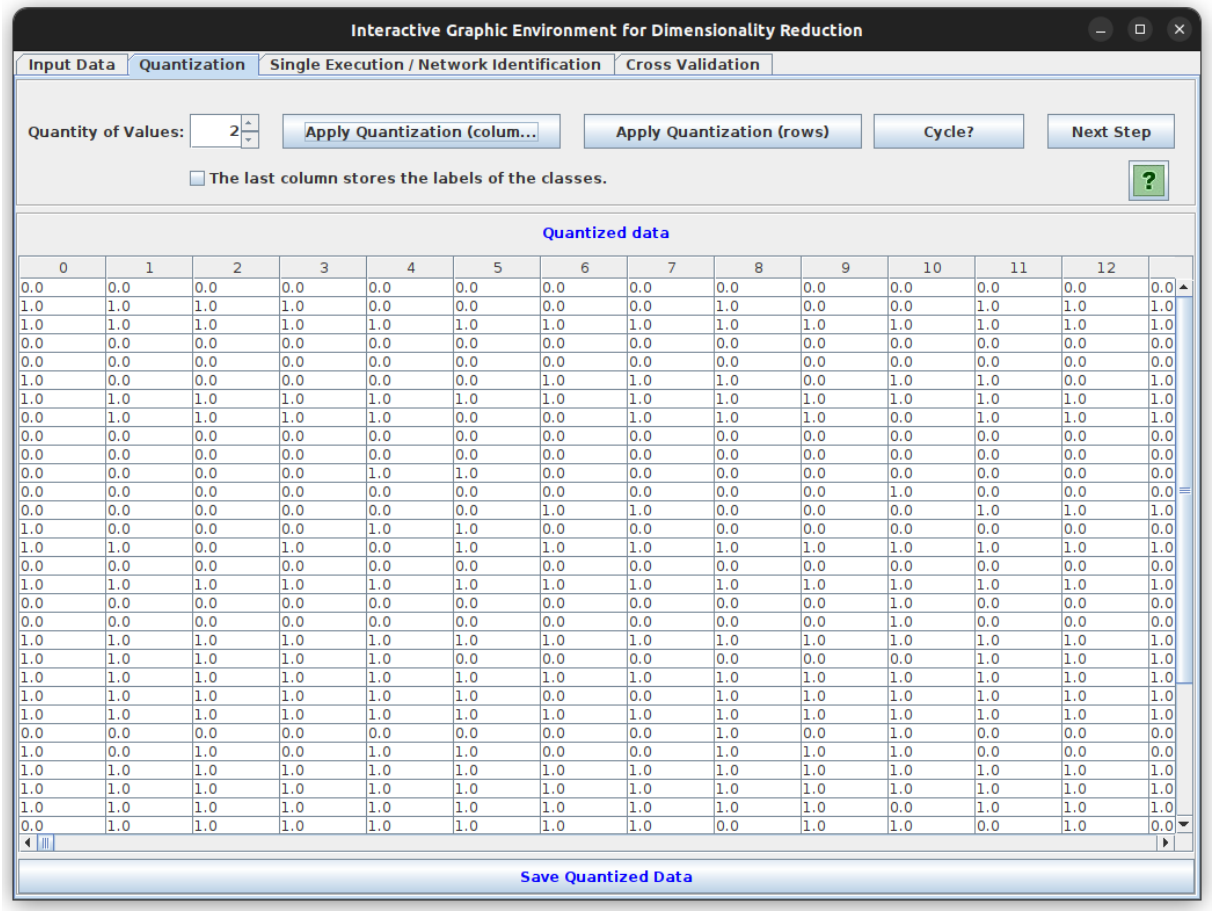


Figure A.2: DimReduction Quantization tab.

- **The last column stores the labels of the classes?:** No. The last column of the dataset does not contain the labels of the classes.

- **Quantization by:** Column. The quantization was performed by column. Since the data was transposed, this means that the quantization process was performed by sample.

The next step was to fill the Single Execution / Network Identification tab to perform the Network Inference, as shown in Figure A.3, with:

- **Criterion Function:** Entropy. The criterion function used was based on the mean conditional entropy. This means that the network inference process evaluates the

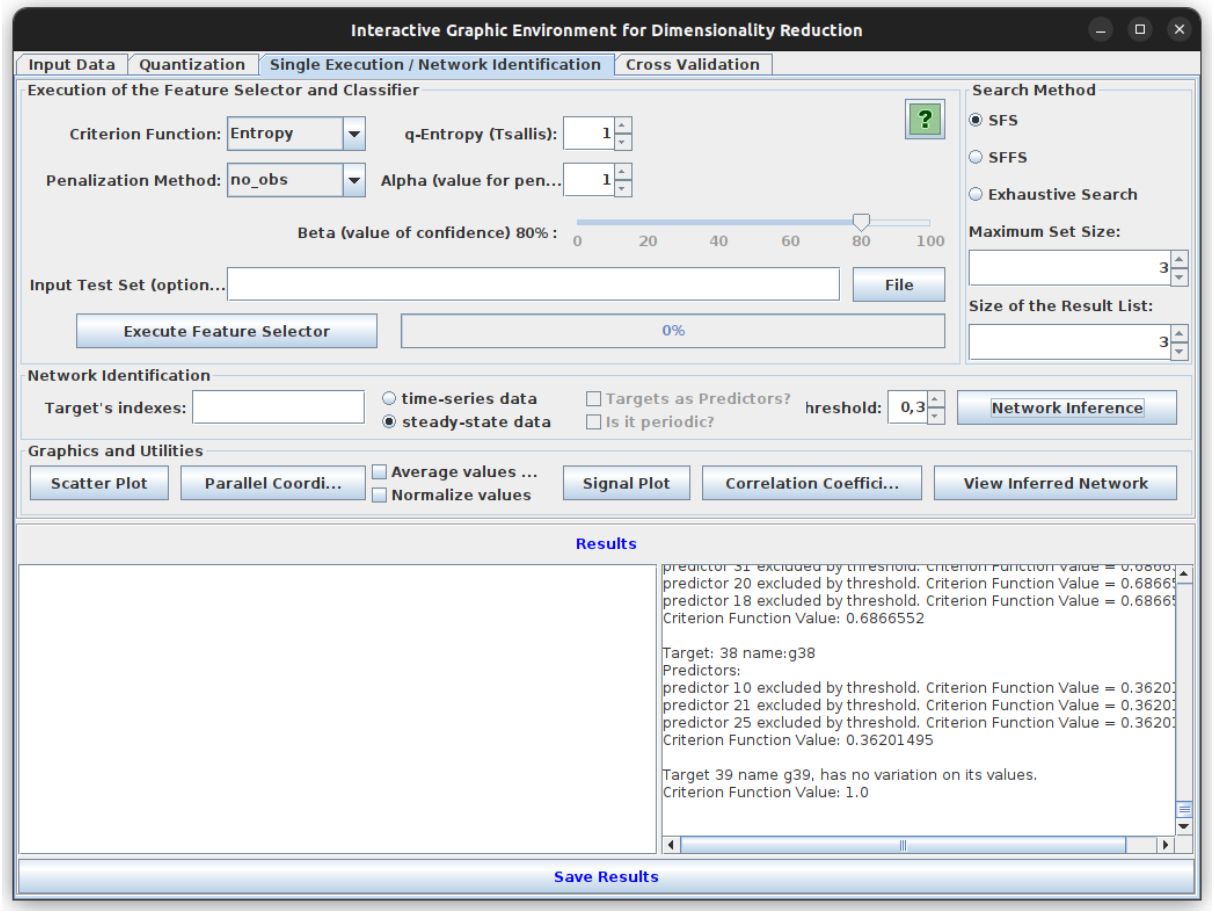


Figure A.3: DimReduction Network Inference tab.

uncertainty or randomness in the data, aiming to identify gene relationships that provide the most information about potential regulatory connections.

- **Penalization Method:** no_obs. Penalize on non-observed instances. This means that the algorithm disadvantages relationships with missing or non-observed values, reducing their importance in the network inference process.

- **q-entropy (Tsallis):** 1. Adjusts non-extensivity in the entropy calculation.

- **Alpha (value for penalty):** 1. Controls the magnitude of penalty imposed on the unobserved instances and the higher the value, the larger the magnitude of penalty.

- **Beta (value of confidence):** 80. Represents the confidence level in the network

inference process, with higher values indicating greater confidence in the identified gene relationships.

- **Search Method:** SFS. The Sequential Forward Selection (SFS) algorithm was used to identify gene relationships, which follow beginning with an empty set of predictors being gradually filled with most relevant predictor at each step.
- **Maximum Set Size:** 3. Denotes the highest number of predictors that one can choose for each target gene in network inference process.
- **Size of the Result List:** 3. Determines the number of top-ranked predictor set displayed as the results.
- **Target's indexes:** Empty. Network inference is applied to all the genes (full network inference). Filling this field would limit this analysis to the target genes specified only.
- **Targets as Predictors?:** No. The features here will be the predictors.
- **Time-series data:** No. The data is steady-state gene expression. All measurements are independently considered and no temporal relationship is assumed between predictors and targets.
- **Is it periodic?:** No. The time series is non periodic in a sense that the last instant of time does not relate with the first instant of time.
- **Threshold:** 0.3. Only the most representative edges will be shown in the graph.

After the inference was finished or the 'View Inferred Network' button was clicked, the graph of Figure A.4, with the network inferred, showed up.

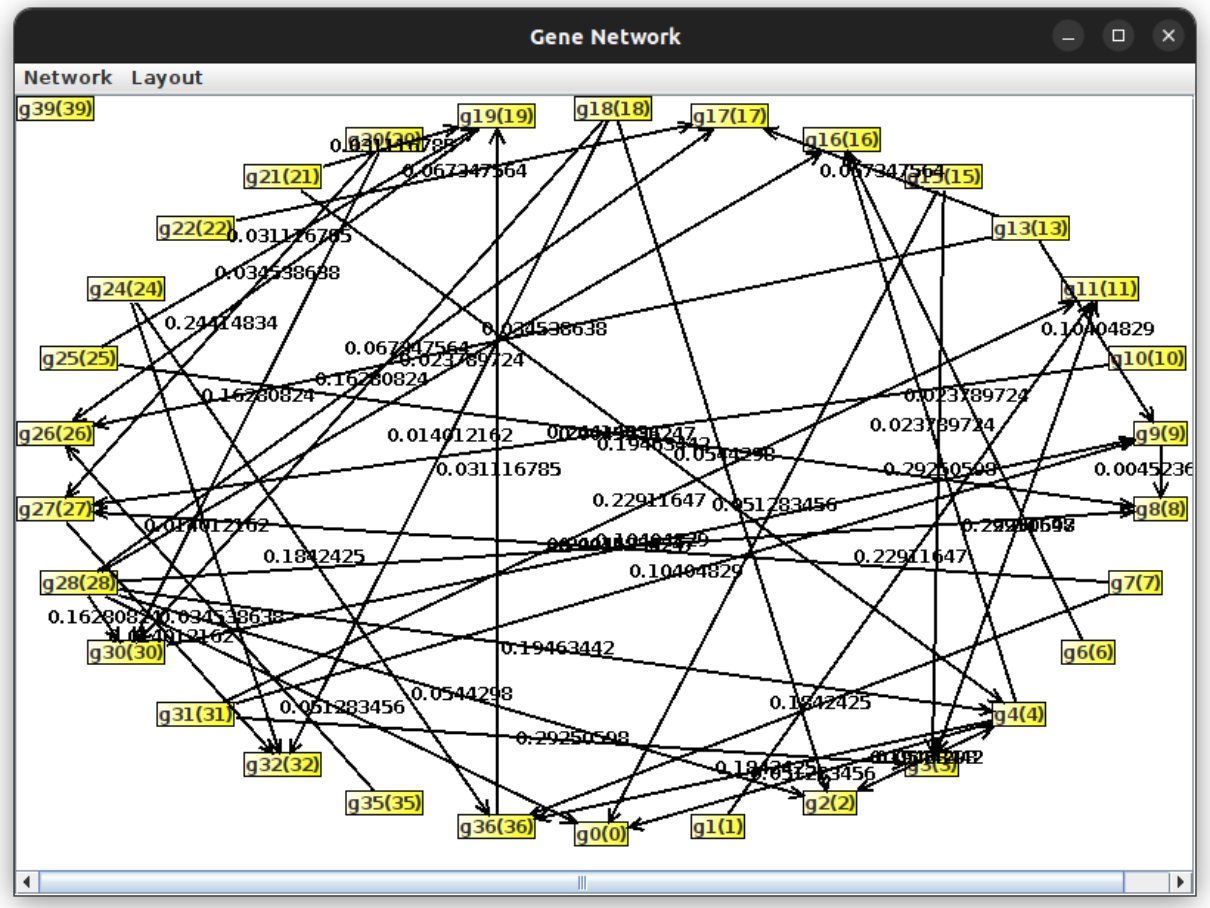


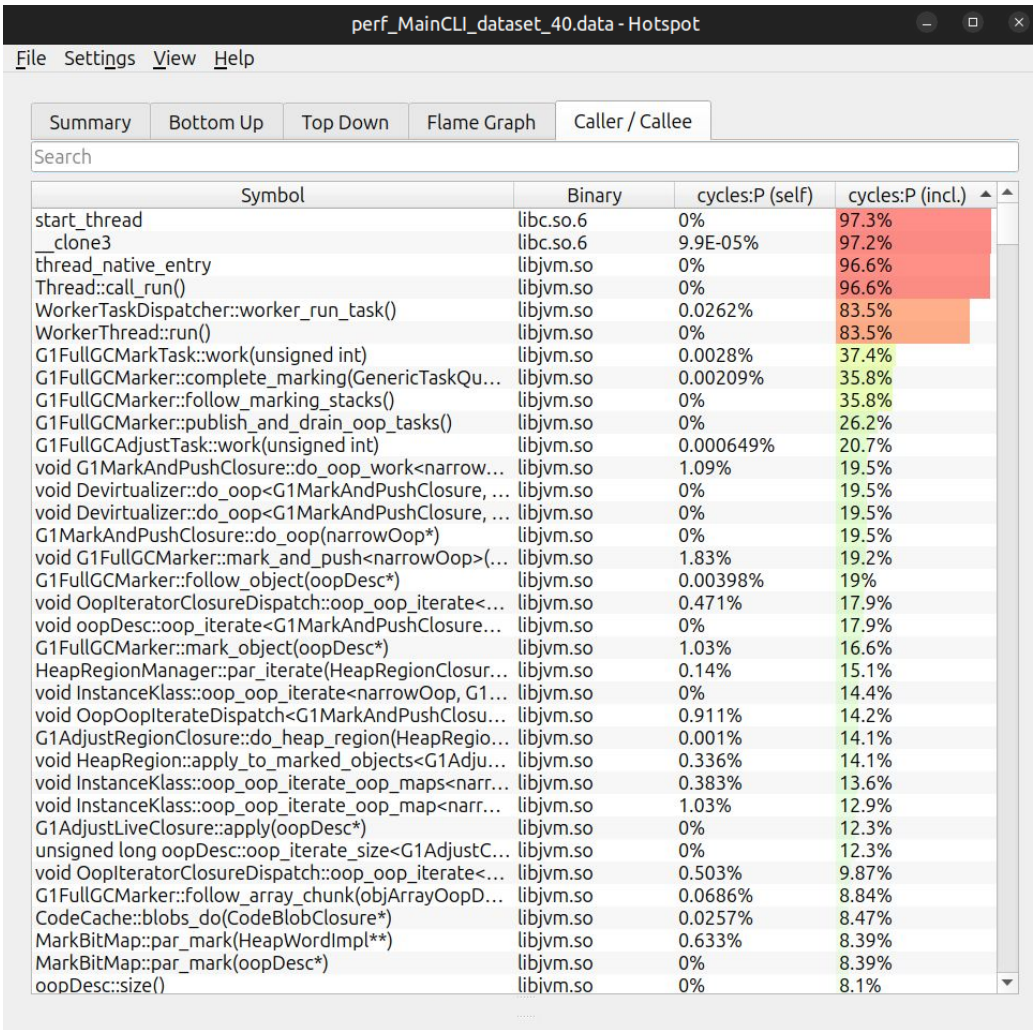
Figure A.4: DimReduction network inference graph for the first 40 genes of the DREAM5 Network 3 dataset.

Appendix B

Profiling Details

This appendix provides supplementary figures from the *Hotspot* profiling analysis, including detailed caller trees and process timelines referenced in Chapter 3.

B.1 Java CLI with 40 Genes



Symbol	Binary	cycles:P (self)	cycles:P (incl.)
start_thread	libc.so.6	0%	97.3%
__clone3	libc.so.6	9.9E-05%	97.2%
thread_native_entry	libjvm.so	0%	96.6%
Thread::call_run()	libjvm.so	0%	96.6%
WorkerTaskDispatcher::worker_run_task()	libjvm.so	0.0262%	83.5%
WorkerThread::run()	libjvm.so	0%	83.5%
G1FullGCMarkTask::work(unsigned int)	libjvm.so	0.0028%	37.4%
G1FullGCMarker::complete_marking(GenericTaskQu…	libjvm.so	0.00209%	35.8%
G1FullGCMarker::follow_marking_stacks()	libjvm.so	0%	35.8%
G1FullGCMarker::publish_and_drain_oop_tasks()	libjvm.so	0%	26.2%
G1FullGCAdjustTask::work(unsigned int)	libjvm.so	0.000649%	20.7%
void G1MarkAndPushClosure::do_oop_work<narrow…	libjvm.so	1.09%	19.5%
void Devirtualizer::do_oop<G1MarkAndPushClosure, …	libjvm.so	0%	19.5%
void Devirtualizer::do_oop<G1MarkAndPushClosure, …	libjvm.so	0%	19.5%
G1MarkAndPushClosure::do_oop(narrowOop*)	libjvm.so	0%	19.5%
void G1FullGCMarker::mark_and_push<narrowOop>(…	libjvm.so	1.83%	19.2%
G1FullGCMarker::follow_object(oopDesc*)	libjvm.so	0.00398%	19%
void OopIteratorClosureDispatch::oop_oop_iterate<…	libjvm.so	0.471%	17.9%
void oopDesc::oop_iterate<G1MarkAndPushClosure…	libjvm.so	0%	17.9%
G1FullGCMarker::mark_object(oopDesc*)	libjvm.so	1.03%	16.6%
HeapRegionManager::par_iterate(HeapRegionClosur…	libjvm.so	0.14%	15.1%
void InstanceKlass::oop_oop_iterate<narrowOop, G1…	libjvm.so	0%	14.4%
void OopOopIterateDispatch<G1MarkAndPushClosu…	libjvm.so	0.911%	14.2%
G1AdjustRegionClosure::do_heap_region(HeapRegio…	libjvm.so	0.001%	14.1%
void HeapRegion::apply_to_marked_objects<G1Adju…	libjvm.so	0.336%	14.1%
void InstanceKlass::oop_oop_iterate_oop_maps<narr…	libjvm.so	0.383%	13.6%
void InstanceKlass::oop_oop_iterate_oop_map<narr…	libjvm.so	1.03%	12.9%
G1AdjustLiveClosure::apply(oopDesc*)	libjvm.so	0%	12.3%
unsigned long oopDesc::oop_iterate_size<G1AdjustC…	libjvm.so	0%	12.3%
void OopIteratorClosureDispatch::oop_oop_iterate<…	libjvm.so	0.503%	9.87%
G1FullGCMarker::follow_array_chunk(objArrayOopD…	libjvm.so	0.0686%	8.84%
CodeCache::blobs_do(CodeBlobClosure*)	libjvm.so	0.0257%	8.47%
MarkBitMap::par_mark(HeapWordImpl**)	libjvm.so	0.633%	8.39%
MarkBitMap::par_mark(oopDesc*)	libjvm.so	0%	8.39%
oopDesc::size()	libjvm.so	0%	8.1%

Figure B.1: Caller Tree Highlighting GC Impact of Java CLI with 40 Genes

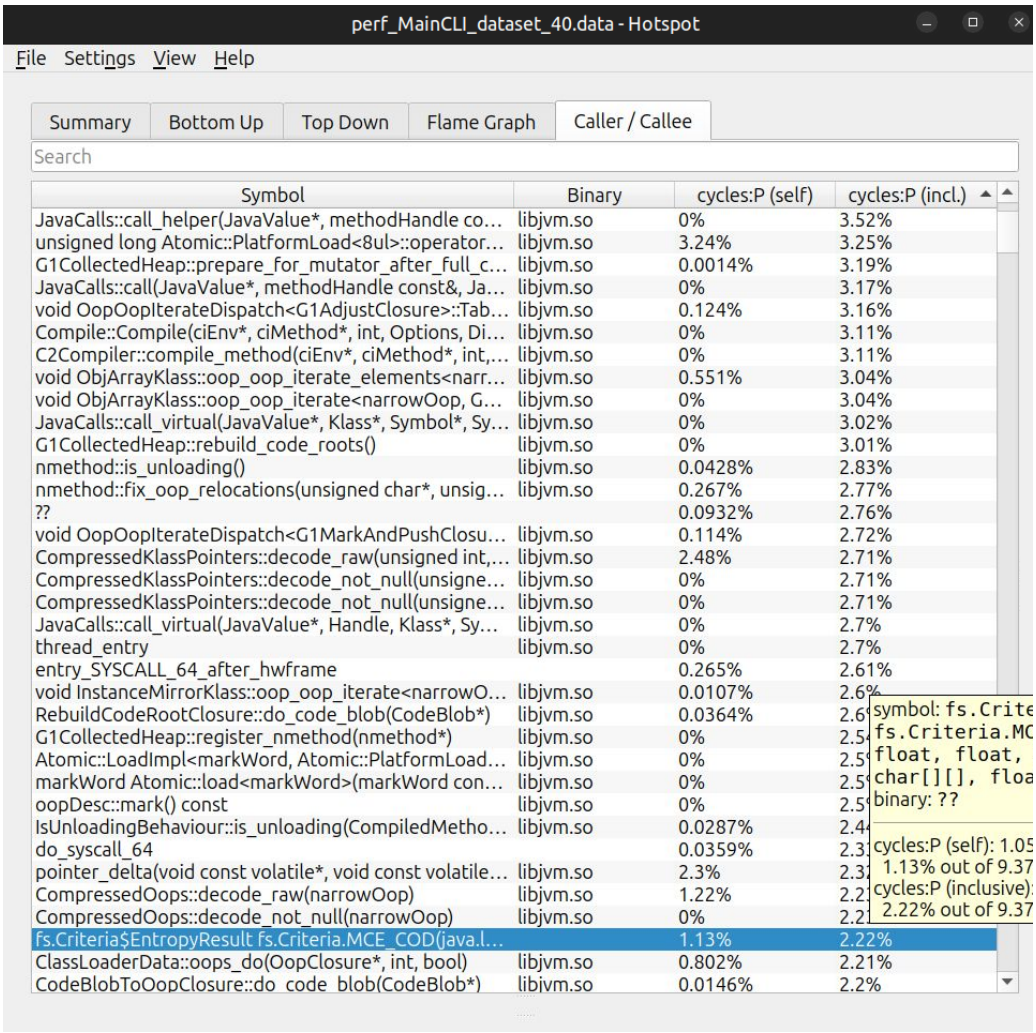


Figure B.2: Caller Tree Focused on MCE_COD of Java CLI with 40 Genes

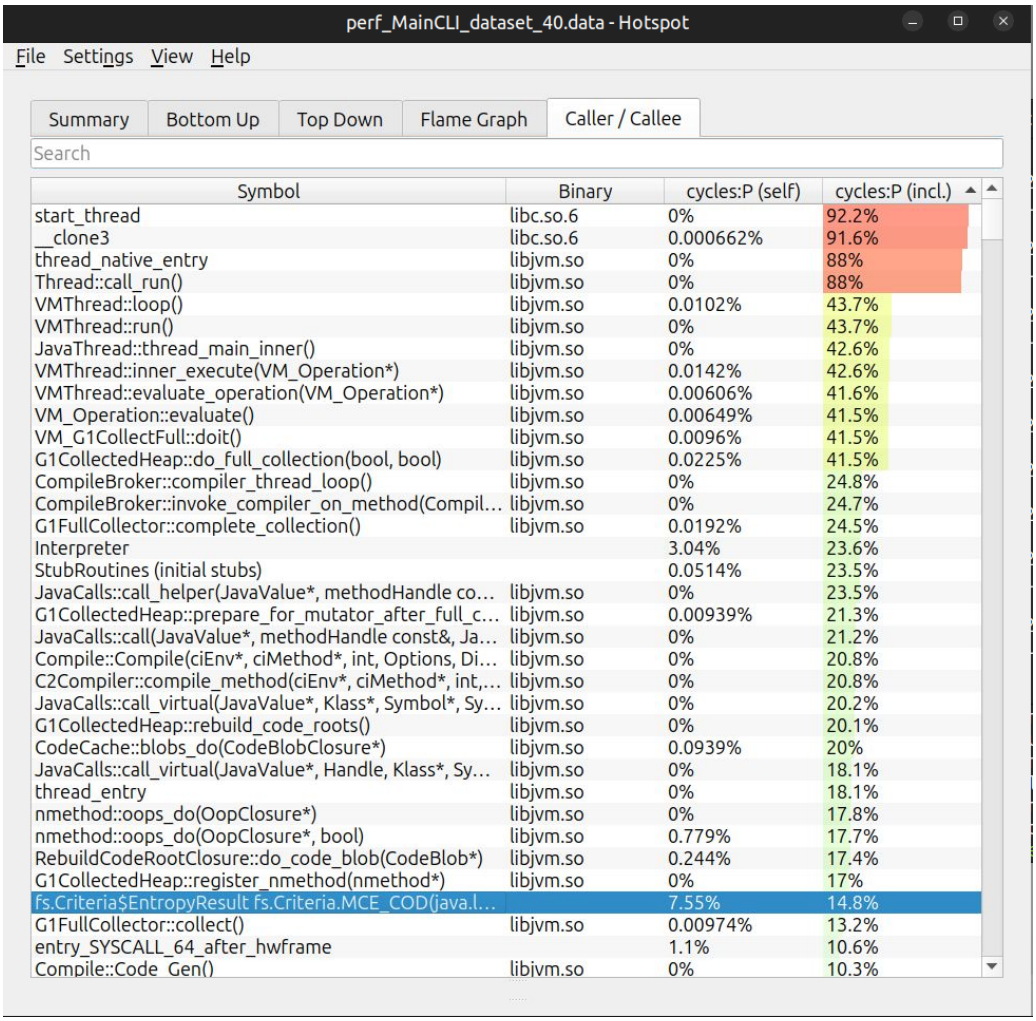


Figure B.3: Filtered Caller Tree for MCE_COD Cost Analysis of Java CLI with 40 Genes

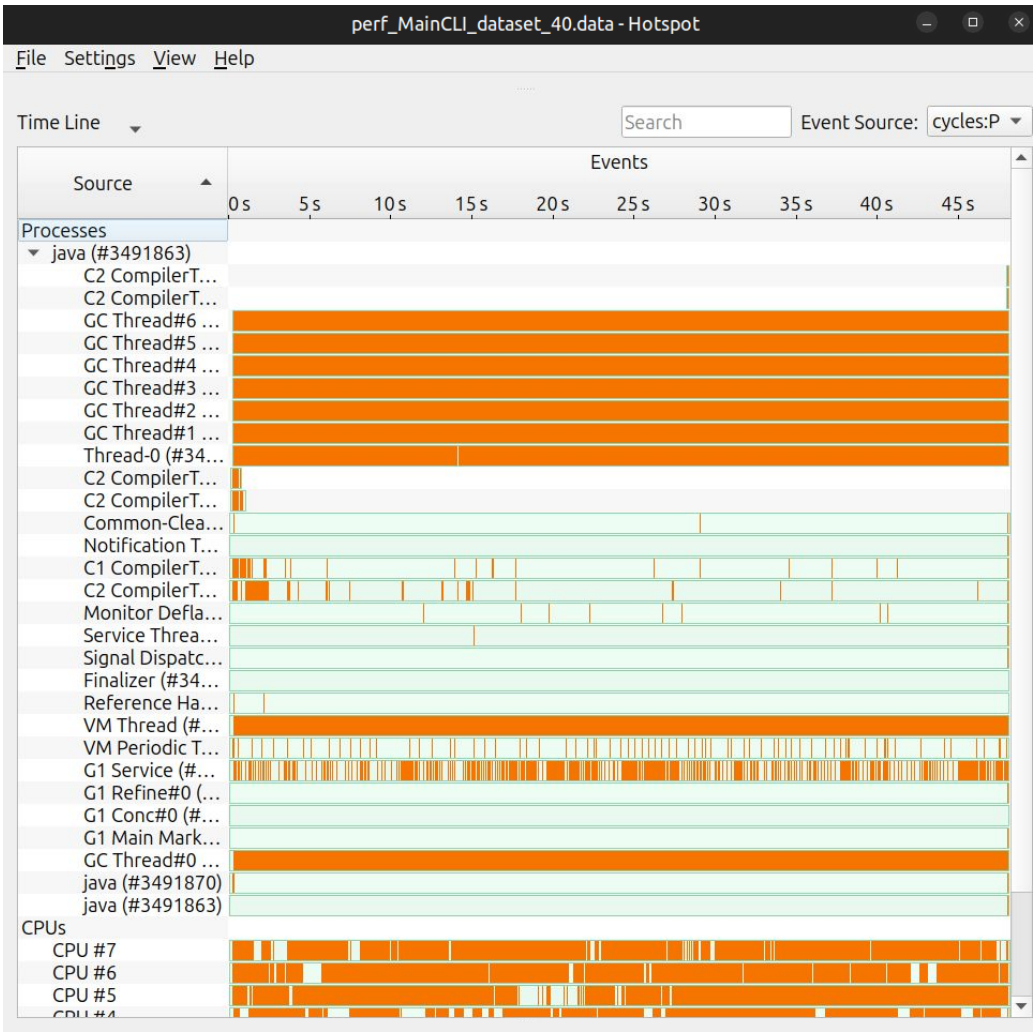


Figure B.4: Process Timeline Showing Persistent GC Thread Activity During Execution of Java CLI with 40 Genes

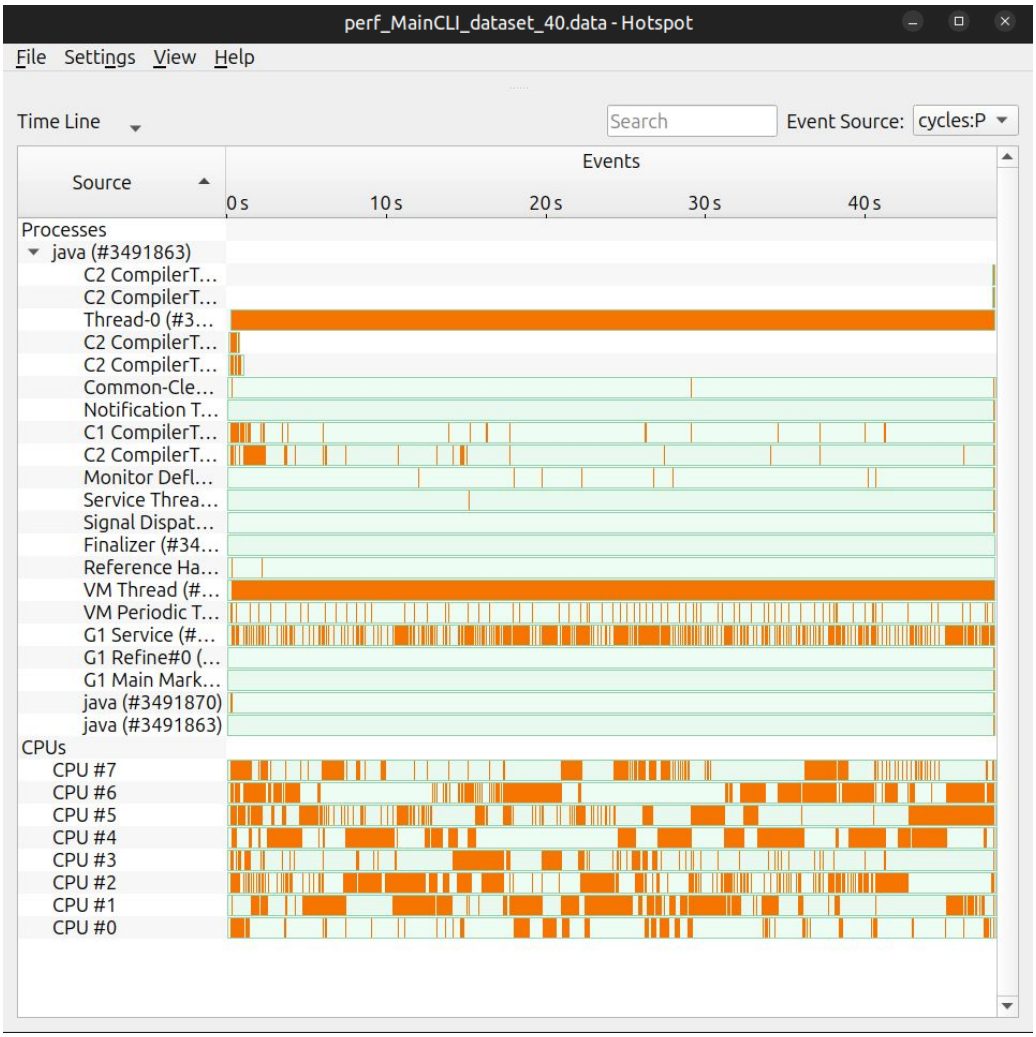


Figure B.5: Filtered Process View Excluding GC Threads for Code-Focused Analysis of Java CLI with 40 Genes

B.2 Python CLI with 40 Genes

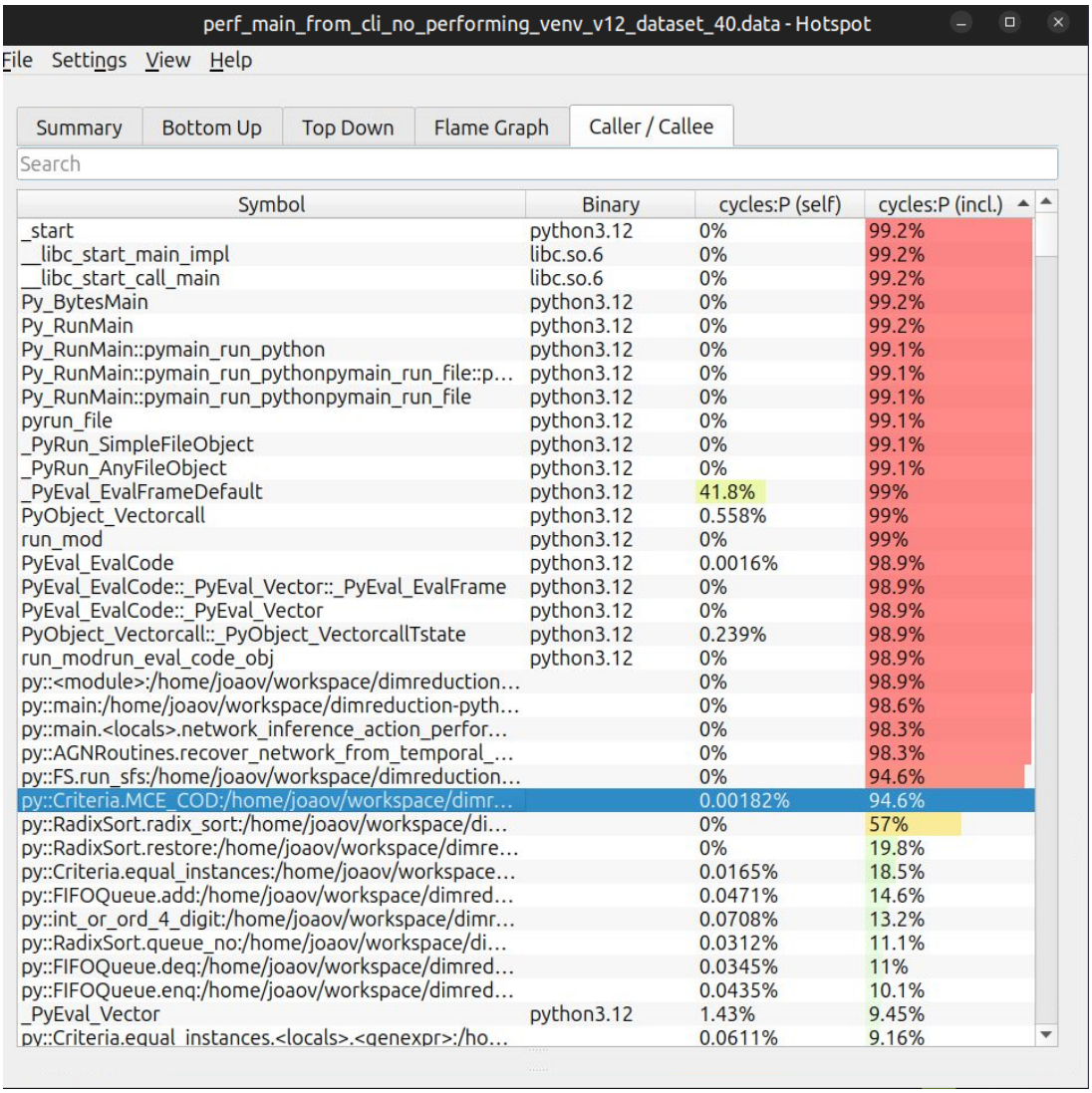
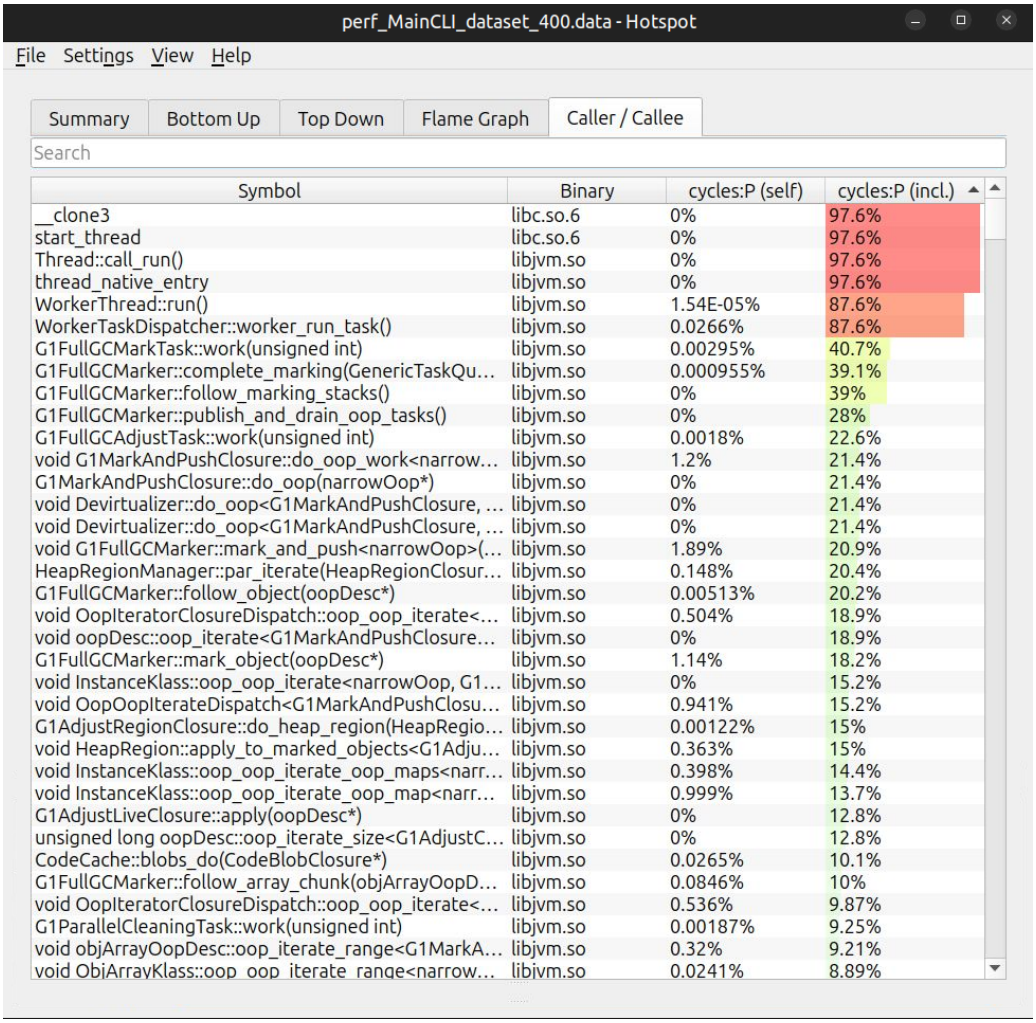


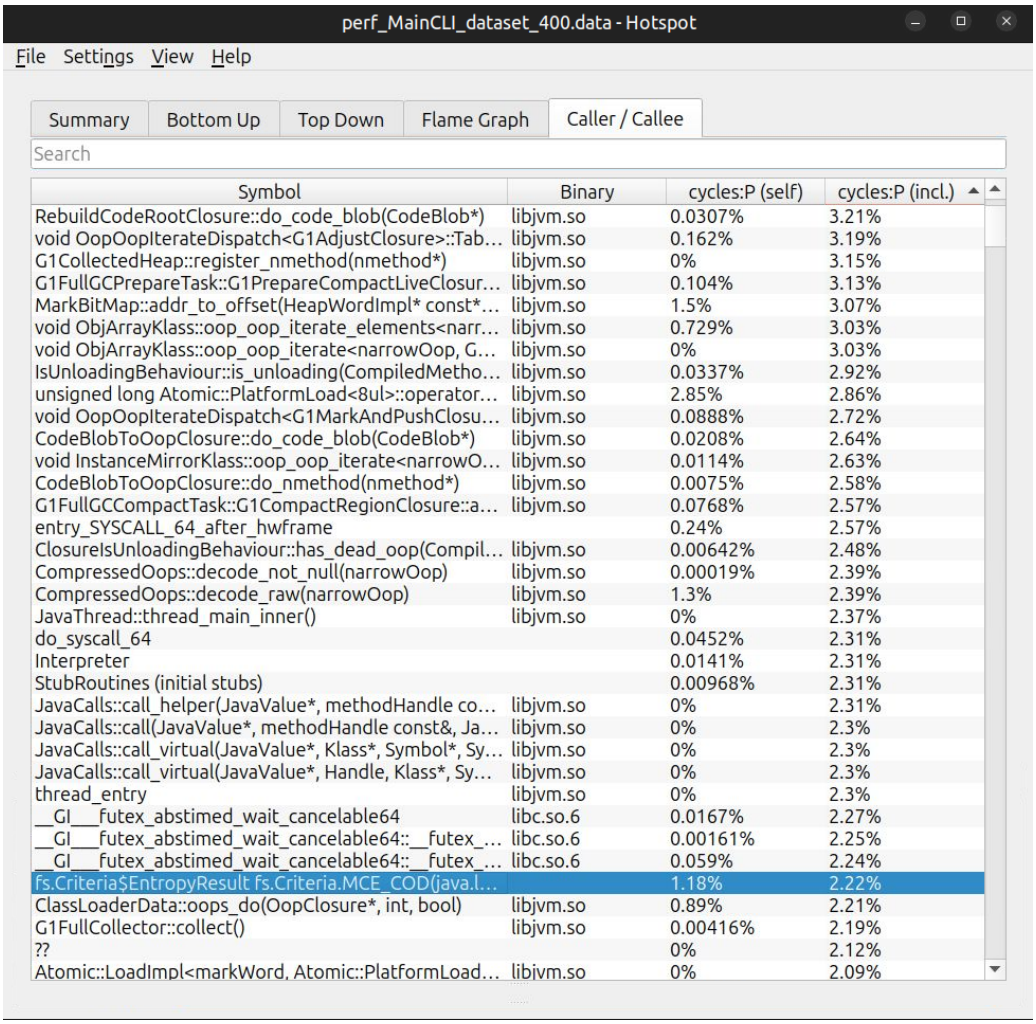
Figure B.6: Caller Tree Showing MCE_COD Dominance of Python CLI with 40 Genes

B.3 Java CLI with 400 Genes



Symbol	Binary	cycles:P (self)	cycles:P (incl.)
__clone3	libc.so.6	0%	97.6%
start_thread	libc.so.6	0%	97.6%
Thread::call_run()	libjvm.so	0%	97.6%
thread_native_entry	libjvm.so	0%	97.6%
WorkerThread::run()	libjvm.so	1.54E-05%	87.6%
WorkerTaskDispatcher::worker_run_task()	libjvm.so	0.0266%	87.6%
G1FullGCMarkTask::work(unsigned int)	libjvm.so	0.00295%	40.7%
G1FullGCMarker::complete_marking(GenericTaskQu…	libjvm.so	0.000955%	39.1%
G1FullGCMarker::follow_marking_stacks()	libjvm.so	0%	39%
G1FullGCMarker::publish_and_drain_oop_tasks()	libjvm.so	0%	28%
G1FullGCAdjustTask::work(unsigned int)	libjvm.so	0.0018%	22.6%
void G1MarkAndPushClosure::do_oop_work<narrow…	libjvm.so	1.2%	21.4%
G1MarkAndPushClosure::do_oop(narrowOop*)	libjvm.so	0%	21.4%
void Devirtualizer::do_oop<G1MarkAndPushClosure, …	libjvm.so	0%	21.4%
void Devirtualizer::do_oop<G1MarkAndPushClosure, …	libjvm.so	0%	21.4%
void G1FullGCMarker::mark_and_push<narrowOop>(…	libjvm.so	1.89%	20.9%
HeapRegionManager::par_iterate(HeapRegionClosur…	libjvm.so	0.148%	20.4%
G1FullGCMarker::follow_object(oopDesc*)	libjvm.so	0.00513%	20.2%
void OopIteratorClosureDispatch::oop_oop_iterate<…	libjvm.so	0.504%	18.9%
void oopDesc::oop_iterate<G1MarkAndPushClosure…	libjvm.so	0%	18.9%
G1FullGCMarker::mark_object(oopDesc*)	libjvm.so	1.14%	18.2%
void InstanceKlass::oop_oop_iterate<narrowOop, G1…	libjvm.so	0%	15.2%
void OopOopIterateDispatch<G1MarkAndPushClosu…	libjvm.so	0.941%	15.2%
G1AdjustRegionClosure::do_heap_region(HeapRegio…	libjvm.so	0.00122%	15%
void HeapRegion::apply_to_marked_objects<G1Adju…	libjvm.so	0.363%	15%
void InstanceKlass::oop_oop_iterate_oop_maps<narr…	libjvm.so	0.398%	14.4%
void InstanceKlass::oop_oop_iterate_oop_map<narr…	libjvm.so	0.999%	13.7%
G1AdjustLiveClosure::apply(oopDesc*)	libjvm.so	0%	12.8%
unsigned long oopDesc::oop_iterate_size<G1AdjustC…	libjvm.so	0%	12.8%
CodeCache::blobs_do(CodeBlobClosure*)	libjvm.so	0.0265%	10.1%
G1FullGCMarker::follow_array_chunk(objArrayOopD…	libjvm.so	0.0846%	10%
void OopIteratorClosureDispatch::oop_oop_iterate<…	libjvm.so	0.536%	9.87%
G1ParallelCleaningTask::work(unsigned int)	libjvm.so	0.00187%	9.25%
void objArrayOopDesc::oop_iterate_range<G1MarkA…	libjvm.so	0.32%	9.21%
void ObjArrayKlass::oop_oop_iterate_range<narrow…	libjvm.so	0.0241%	8.89%

Figure B.7: Caller Tree Highlighting GC Impact of Java CLI with 400 Genes



Symbol	Binary	cycles:P (self)	cycles:P (incl.)
RebuildCodeRootClosure::do_code_blob(CodeBlob*)	libjvm.so	0.0307%	3.21%
void OopOopIterateDispatch<G1AdjustClosure>::Tab…	libjvm.so	0.162%	3.19%
G1CollectedHeap::register_nmethod(nmethod*)	libjvm.so	0%	3.15%
G1FullGCPrepareTask::G1PrepareCompactLiveClosur…	libjvm.so	0.104%	3.13%
MarkBitMap::addr_to_offset(HeapWordImpl* const*…	libjvm.so	1.5%	3.07%
void ObjArrayKlass::oop_oop_iterate_elements<narr…	libjvm.so	0.729%	3.03%
void ObjArrayKlass::oop_oop_iterate<narrowOop, G…	libjvm.so	0%	3.03%
IsUnloadingBehaviour::is_unloading(CompiledMetho…	libjvm.so	0.0337%	2.92%
unsigned long Atomic::PlatformLoad<8ul>::operator…	libjvm.so	2.85%	2.86%
void OopOopIterateDispatch<G1MarkAndPushClosu…	libjvm.so	0.0888%	2.72%
CodeBlobToOopClosure::do_code_blob(CodeBlob*)	libjvm.so	0.0208%	2.64%
void InstanceMirrorKlass::oop_oop_iterate<narrowO…	libjvm.so	0.0114%	2.63%
CodeBlobToOopClosure::do_nmethod(nmethod*)	libjvm.so	0.0075%	2.58%
G1FullGCCompactTask::G1CompactRegionClosure::a…	libjvm.so	0.0768%	2.57%
entry_SYSCALL_64_after_hwframe		0.24%	2.57%
ClosureIsUnloadingBehaviour::has_dead_oop(Compil…	libjvm.so	0.00642%	2.48%
CompressedOops::decode_not_null(narrowOop)	libjvm.so	0.00019%	2.39%
CompressedOops::decode_raw(narrowOop)	libjvm.so	1.3%	2.39%
JavaThread::thread_main_inner()	libjvm.so	0%	2.37%
do_syscall_64		0.0452%	2.31%
Interpreter		0.0141%	2.31%
StubRoutines (initial stubs)		0.00968%	2.31%
JavaCalls::call_helper(JavaValue*, methodHandle co…	libjvm.so	0%	2.31%
JavaCalls::call(JavaValue*, methodHandle const&, Ja…	libjvm.so	0%	2.3%
JavaCalls::call_virtual(JavaValue*, Klass*, Symbol*, Sy…	libjvm.so	0%	2.3%
JavaCalls::call_virtual(JavaValue*, Handle, Klass*, Sy…	libjvm.so	0%	2.3%
thread_entry	libjvm.so	0%	2.3%
__GI___futex_abstimed_wait_cancelable64	libc.so.6	0.0167%	2.27%
__GI___futex_abstimed_wait_cancelable64::__futex_…	libc.so.6	0.00161%	2.25%
__GI___futex_abstimed_wait_cancelable64::__futex_…	libc.so.6	0.059%	2.24%
fs.Criteria$EntropyResult fs.Criteria.MCE_COD(java.l…		1.18%	2.22%
ClassLoaderData::oops_do(OopClosure*, int, bool)	libjvm.so	0.89%	2.21%
G1FullCollector::collect()	libjvm.so	0.00416%	2.19%
??		0%	2.12%
Atomic::LoadImpl<markWord, Atomic::PlatformLoad…	libjvm.so	0%	2.09%

Figure B.8: Caller Tree Focused on MCE_COD of Java CLI with 400 Genes

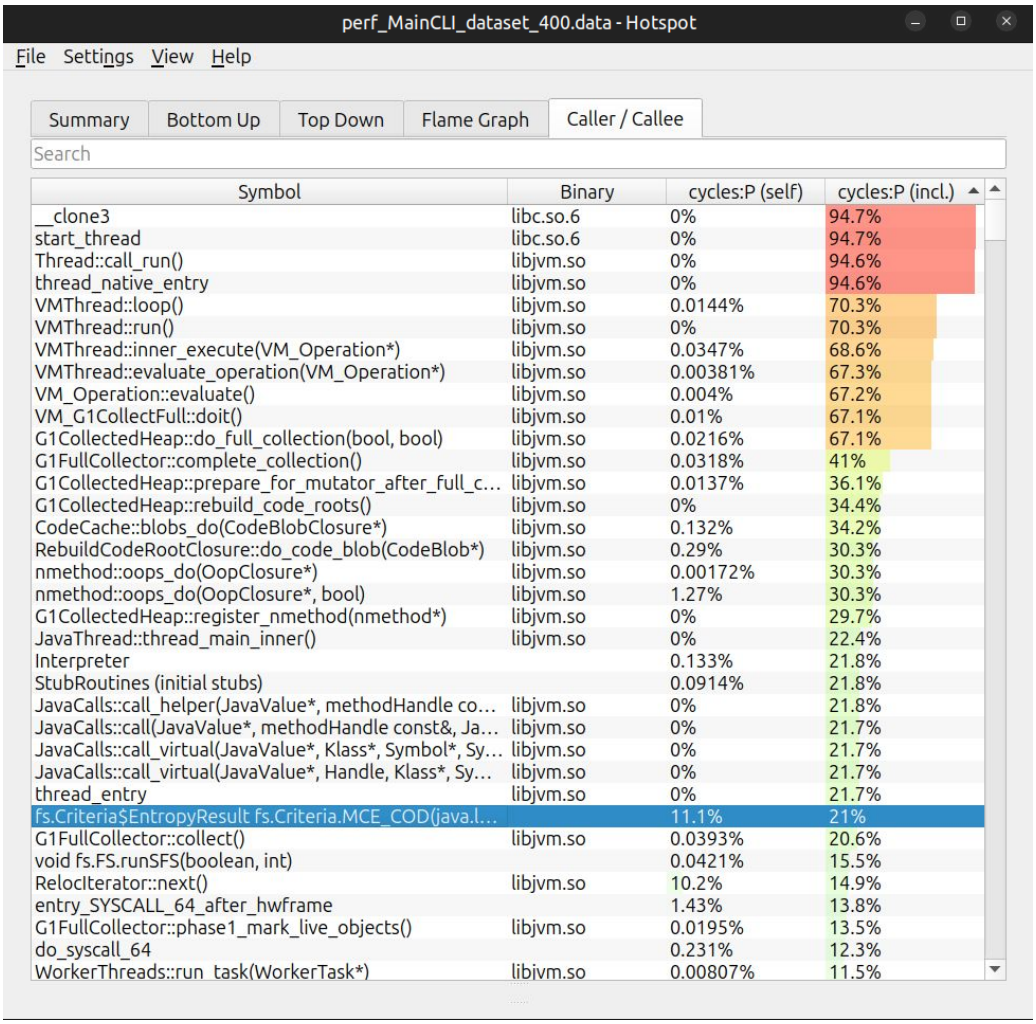


Figure B.9: Filtered Caller Tree for MCE_COD Cost Analysis of Java CLI with 400 Genes

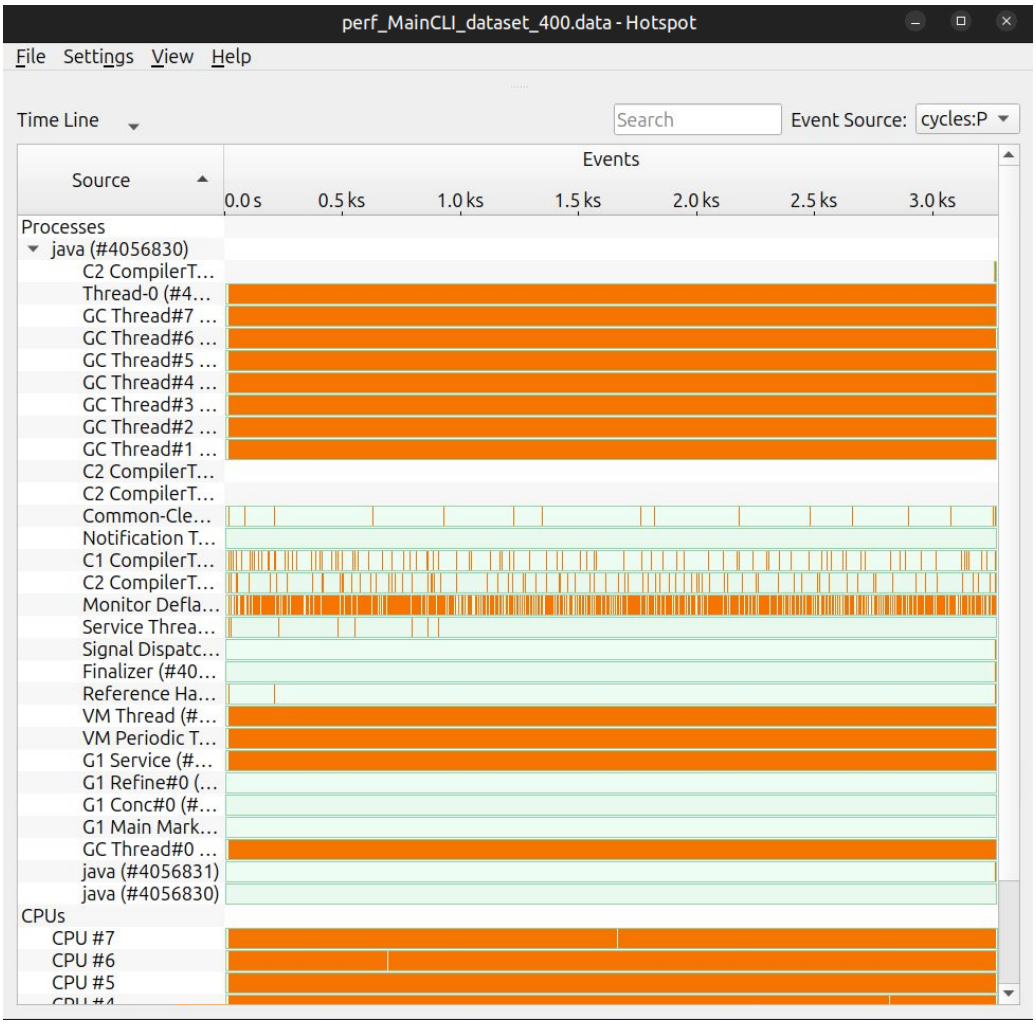


Figure B.10: Process Timeline Showing Persistent GC Thread Activity During Execution of Java CLI with 400 Genes

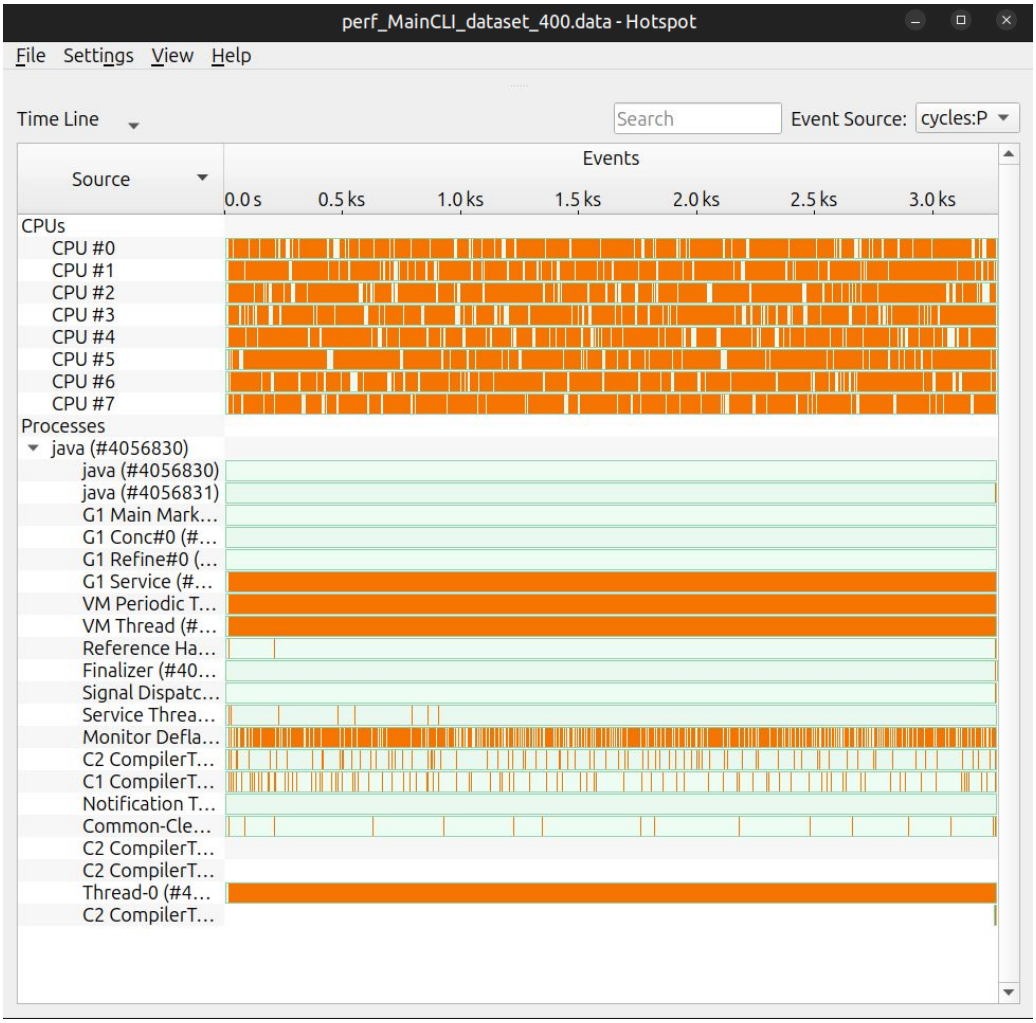


Figure B.11: Filtered Process View Excluding GC Threads for Code-Focused Analysis of Java CLI with 400 Genes

Appendix C

Configuration Files for CLI versions

```
OUTPUT_FOLDER=./results

INPUT_FILE_PATH=/path/to/processed_dataset_dream5_40.csv
# ----------------------------------------

ARE_COLUMNS_DESCRIPTIVE=true
ARE_TITLES_ON_FIRST_COLUMN=false
TRANSPOSE_MATRIX=true

# Quantization
QUANTIZATION_VALUE=2 # at least 1
HAS_LABELS_CLASSES_ON_LAST_COLUMN=false
LOOK_FOR_CYCLES=false
APPLY_QUANTIZATION_TYPE=1 # 1 for columns, 2 for rows, 0 for do
   ↪ not apply
SAVE_QUANTIZED_DATA=true

# FeatureSelection
EXECUTE_FEATURE_SELECTION=false
CRITERION_FUNCTION_FEATURE_SELECTION=0
Q_ENTROPY_FEATURE_SELECTION=1
PENALIZATION_METHOD_FEATURE_SELECTION=0
```

```
ALPHA_FEATURE_SELECTION=1
BETA_FEATURE_SELECTION=80
INPUT_TEST_SET_FEATURE_SELECTION=
SEARCH_METHOD_FEATURE_SELECTION=1 # 1: SFS, 2: Exhaustive Search
    ↪ , 3: SFFS
SAVE_FINAL_DATA=true

# NetworkInference
TARGET_INDEXES=
# TARGET_INDEXES="0 1 2 3"
TARGETS_AS_PREDICTORS=
TIME_SERIES_DATA=false
IS_IT_PERIODIC=
THRESHOLD=0.3

# ---------------------------------------

VERBOSITY_LEVEL=0
```

Listing C.1: Sample .ENV file for Java and Python CLI versions

```
OUTPUT_FOLDER=./results

INPUT_FILE_PATH=/path/to/input_data/geneci/DREAM5/EXP_DimReduction/
    ↪ net3_exp.csv
TF_FILE_PATH=/path/to/Network3_transcription_factors.tsv
# ---------------------------------------

ARE_COLUMNS_DESCRIPTIVE=false
ARE_TITLES_ON_FIRST_COLUMN=true
TRANSPOSE_MATRIX=false

# Quantization
QUANTIZATION_VALUE=2 # at least 1
HAS_LABELS_CLASSES_ON_LAST_COLUMN=false
```

```
LOOK_FOR_CYCLES=false
APPLY_QUANTIZATION_TYPE=1 # 1 for columns, 2 for rows, 0 for do not
    ↪ apply
SAVE_QUANTIZED_DATA=true

# FeatureSelection
EXECUTE_FEATURE_SELECTION=false
CRITERION_FUNCTION_FEATURE_SELECTION=0
Q_ENTROPY_FEATURE_SELECTION=1
PENALIZATION_METHOD_FEATURE_SELECTION=0
ALPHA_FEATURE_SELECTION=1
BETA_FEATURE_SELECTION=80
INPUT_TEST_SET_FEATURE_SELECTION=
SEARCH_METHOD_FEATURE_SELECTION=1 # 1: SFS, 2: Exhaustive Search, 3:
    ↪  SFFS
SAVE_FINAL_DATA=true
SAVE_FINAL_WEIGHT_DATA=true

# NetworkInference
TARGET_INDEXES=
# TARGET_INDEXES="0 1 2 3"
TARGETS_AS_PREDICTORS=
TIME_SERIES_DATA=false
IS_IT_PERIODIC=
THRESHOLD=1.0

# ----------------------------------------
NUMBER_OF_THREADS=1
THREAD_DISTRIBUTION=demain

ENABLE_MANUAL_GC=false
VERBOSITY_LEVEL=0
```

Listing C.2: Sample .ENV file for comparison experiments

Appendix D

Running CLI versions

```
javac -d out/production/java-dimreduction \
      -cp "./lib/*:./out/production/java-dimreduction" \
      src/**/*.java;
cp -r src/img out/production/java-dimreduction/;
java -cp ./lib/*:./out/production/java-dimreduction fs.MainCLI;
```

Listing D.1: Commands to compile a run the Java CLI version.

```
diff --from-file ../saving_files/40genes/0000gui-quantized_data.txt
   ↪ \
      results/*/*quantized_data.txt \
      -sq;
diff --from-file ../saving_files/40genes/0000gui-final_data.txt \
      results/*/*final_data.txt \
      -sq;
```

Listing D.2: Comparing results from different DimReduction versions.

```
python main_from_cli_no_performing.py;
```

Listing D.3: Running the Python CLI version.

Appendix E

System Monitoring Individual Execution Plots

This appendix contains the detailed, individual execution plots from the system monitoring analysis conducted in Chapter 3. These graphs provide a granular view of CPU usage, system load, and memory utilization for each test run.

E.1 Java Performance (40 Genes)

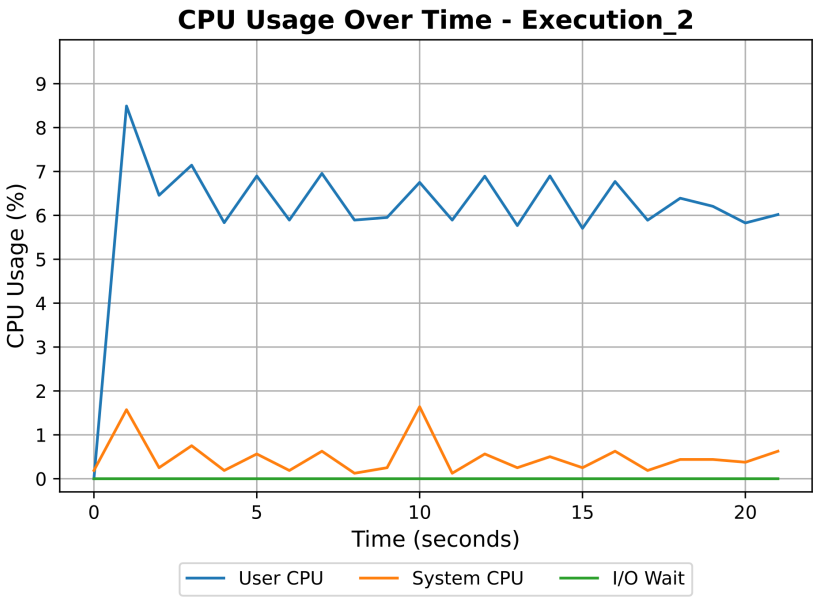


Figure E.1: CPU Usage During Execution 2 (Java CLI with 40 Genes)

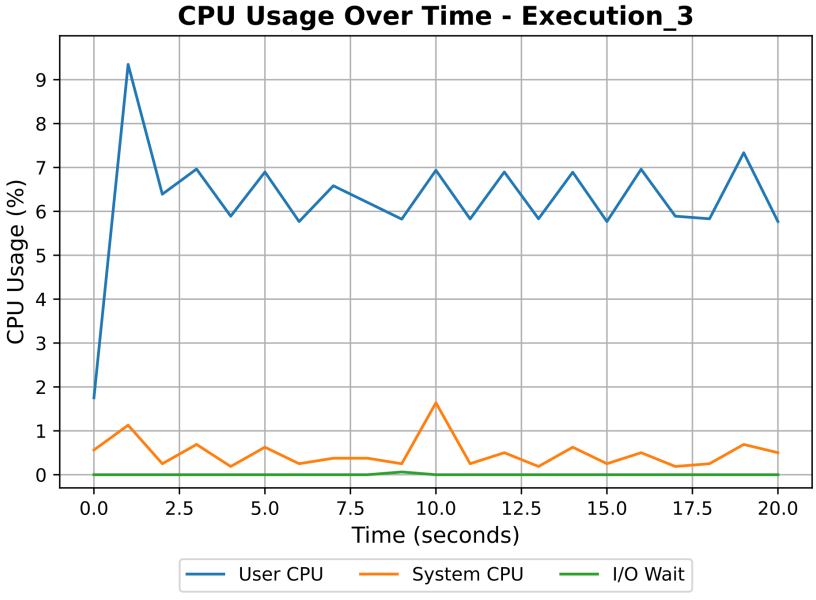


Figure E.2: CPU Usage During Execution 3 (Java CLI with 40 Genes)

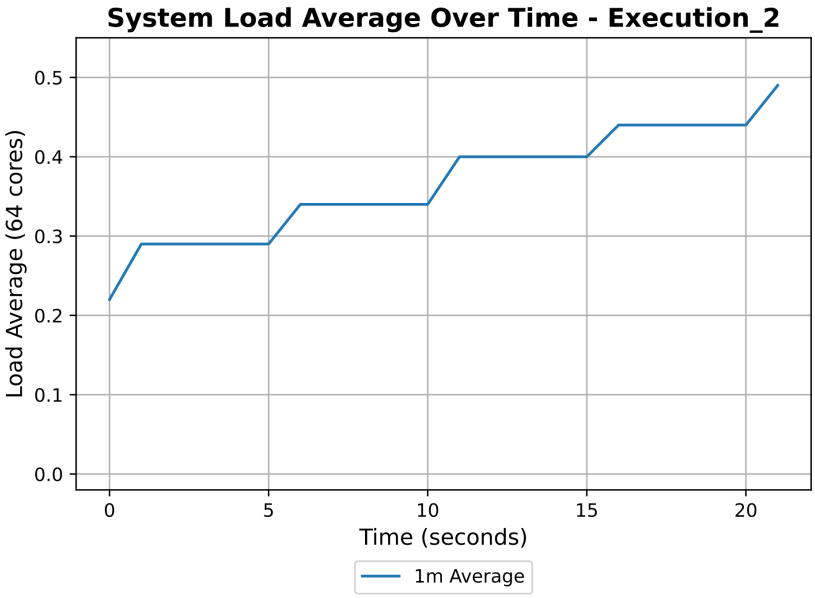


Figure E.3: Load Average During Execution 2 (Java CLI with 40 Genes)

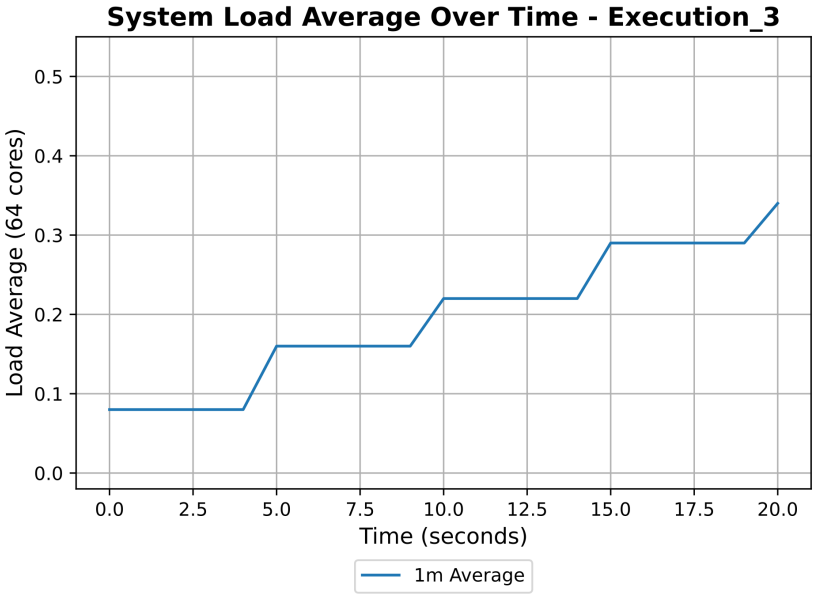


Figure E.4: Load Average During Execution 3 (Java CLI with 40 Genes)

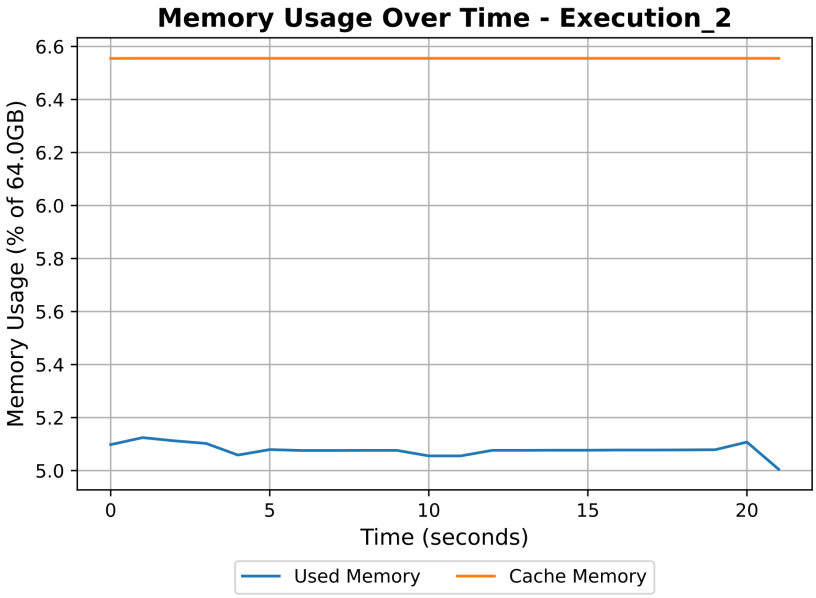


Figure E.5: Memory Utilization During Execution 2 (Java CLI with 40 Genes)

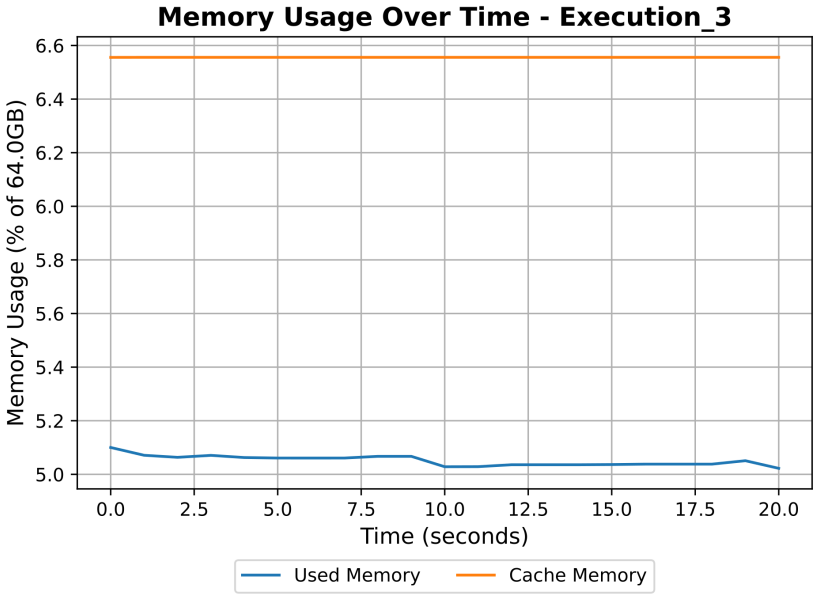


Figure E.6: Memory Utilization During Execution 3 (Java CLI with 40 Genes)

E.2 Python Performance (40 Genes)

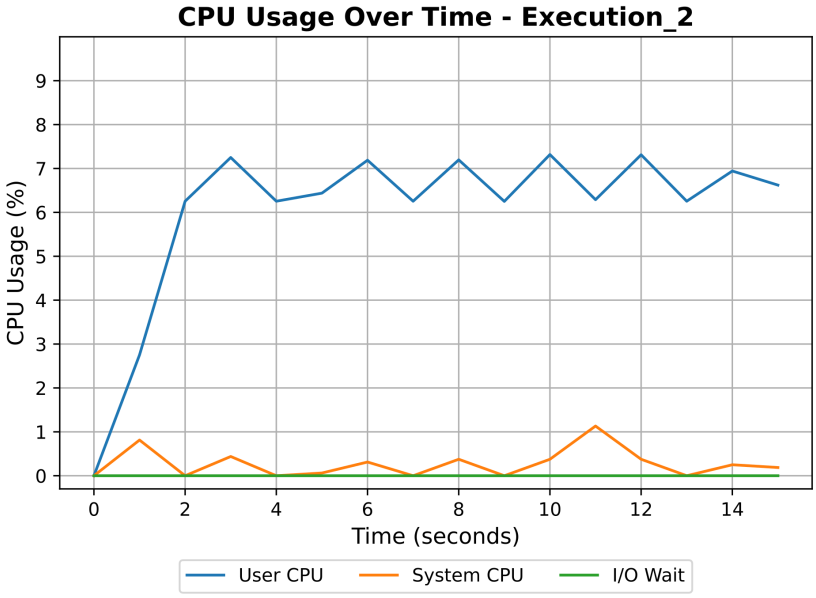


Figure E.7: CPU Usage During Execution 2 (Python CLI with 40 Genes)

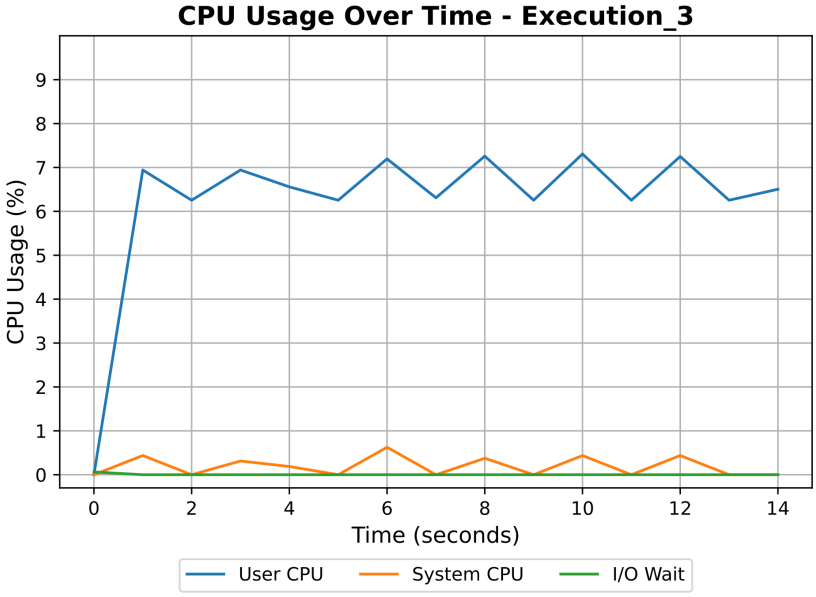


Figure E.8: CPU Usage During Execution 3 (Python CLI with 40 Genes)

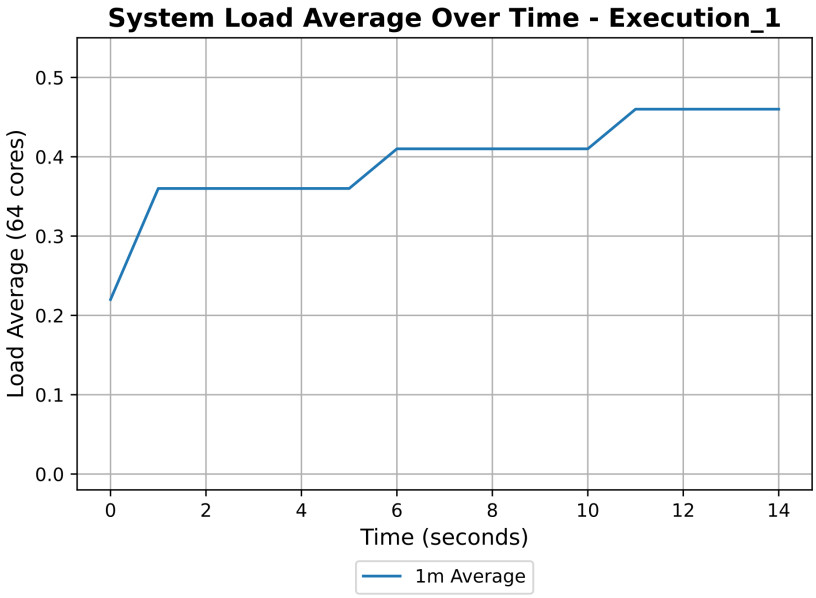


Figure E.9: Load Average During Execution 1 (Python CLI with 40 Genes)

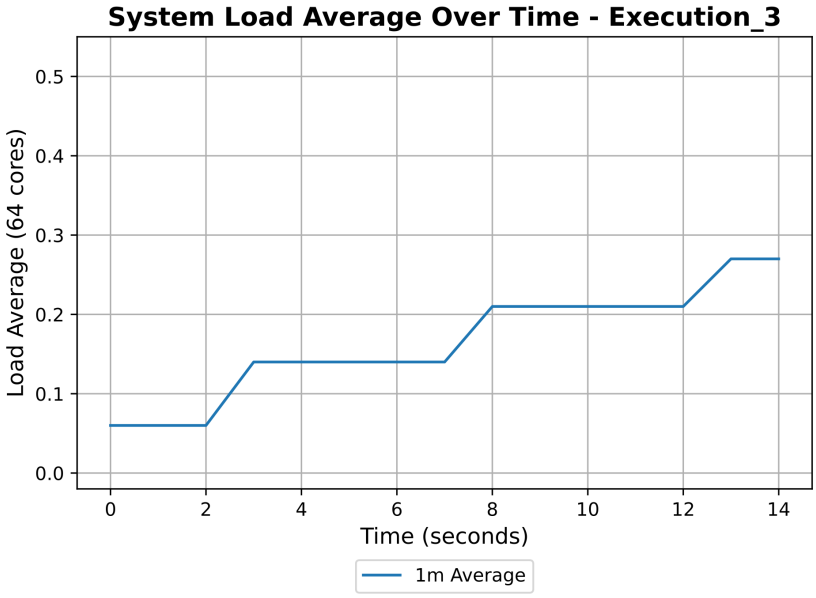


Figure E.10: Load Average During Execution 3 (Python CLI with 40 Genes)

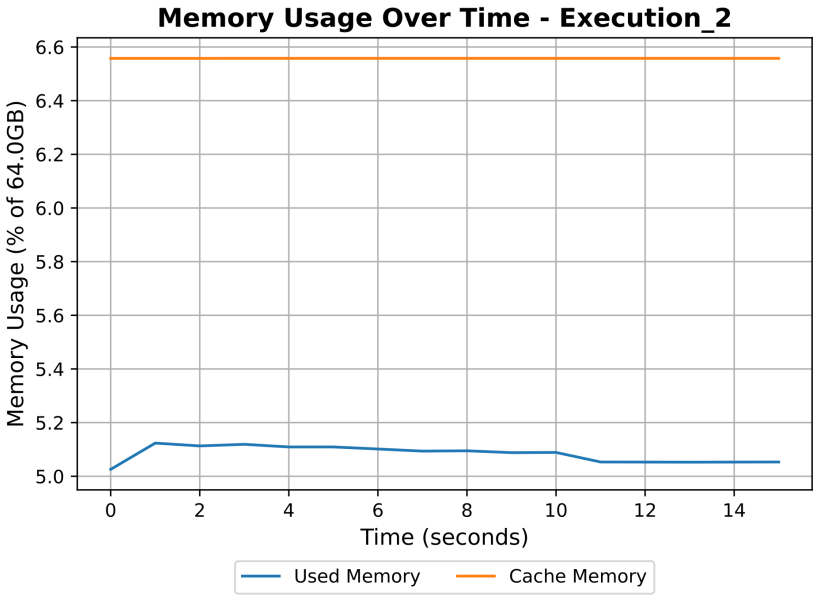


Figure E.11: Memory Utilization During Execution 2 (Python CLI with 40 Genes)

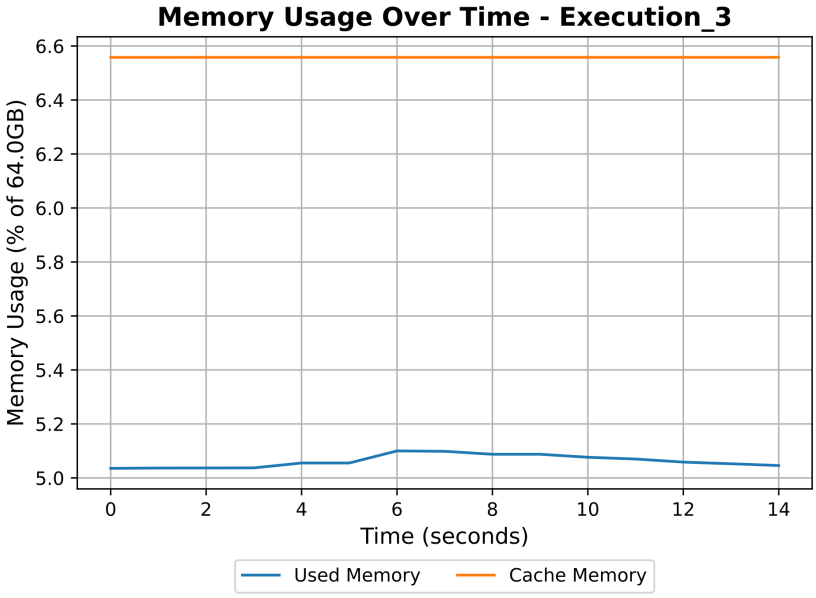


Figure E.12: Memory Utilization During Execution 3 (Python CLI with 40 Genes)

Appendix F

Analysis Tools Configurations

To conduct a performance analysis using *Perf* and *Hotspot*, several configuration steps were needed, as next outlined for a Linux system with an `apt`-based package manager.

Tools Installation Install the required profiling tools:

```
# Install perf from Linux tools
sudo apt install linux-tools-generic

# Install Hotspot (Flame Graph viewer)
sudo apt install hotspot
```

System Configuration Adjust kernel parameters in order to allow proper profiling:

```
# Give access to the non-root users to the kernel symbols
sudo sh -c 'echo 0 > /proc/sys/kernel/kptr_restrict'

# Less restrict security for events
sudo sh -c 'echo 1 > /proc/sys/kernel/perf_event_paranoid'
```

Java Configuration Profiling of Java application requires some JVM flags:

```
java -XX:+UnlockDiagnosticVMOptions \
     -XX:+DumpPerfMapAtExit \
     -XX:+PreserveFramePointer \
     -cp ./lib/*:./out/production/java-dimreduction \
     fs.MainCLI
```

The meaning of these flags is the following:

- `-XX:+UnlockDiagnosticVMOptions` – Allows to access diagnostic JVM options
- `-XX:+DumpPerfMapAtExit` – Generates a mapping file which helps *Perf* convert memory addresses to Java method names.
- `-XX:+PreserveFramePointer` – Keeps the frame pointer in call stack, which assists *Perf* in creating precise stack traces.

Python Configuration The profiling of Python 3.12+ applications was done with:

```
python -X perf main_from_cli_no_performing.py
```

The `-X perf` flag enables the inbuilt support for *Perf* profiling that was added to Python starting 3.12.

Before the profiling could be run, it had to be confirmed that Python get compiled with the necessary support:

```
# Make sure trampoline support for perf is on.
python -m sysconfig | grep HAVE_PERF_TRAMPOLINE


# Ensure that frame pointers will be maintained
python -m sysconfig | grep 'no-omit-frame-pointer'
```

These checks guarantee that Python was constructed with adequate support to provide exact profiling information to the *Perf* tool.

Data Collection After a 1st test run with Hotspot with default settings, a more defined data collection can be achieved by executing a particular *Perf* record command:

```
perf record -o ${output_dir}/perf_${script}_dataset_${DATASET_NAME}.data \
    --call-graph dwarf \
    --aio \
    --sample-cpu \
    --mmap-pages 16M \
    <command-to-run-application>
```

The meaning of the options used above is as follows:

- `-o` – It identifies the output file for the gathered data.
- `-call-graph dwarf` – Utilizes debugging information for DWARF in order to reconstruct the call graph with better results than the frame pointer default method.
- `-aio` – Allows asynchronous I/O for better performance while collecting data.
- `-sample-cpu` – Tracks data in per-CPU form for further analysis.
- `-mmap-pages 16M` – Creates 16MB of memory for the *Perf* buffer, which prevents data loss during collection.

The subsequently collected data can then analyzed using *Hotspot* for interactive visualization of flame graphs and other performance metrics, by running:

```
hotspot ${output_dir}/perf_${script}_dataset_${DATASET_NAME}.data
```