A Million Proteins per Second: Scaling a Protein Comparison Tool for Biological Breakthroughs

Master's Thesis

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
High Performance Computing Group

Advisors and Examiners:
Prof. Dr. Florina M. Ciorba[1]
Dr. Joana Maria Soares Pereira[2]
Supervisor: Dr. Osman Seckin Simsek[1]

[1] Department of Mathematics and Computer Science, University of Basel
[2] Biozentrum, University of Basel

Reto Krummenacher
reto.krummenacher@unibas.ch
03-054-327

August 30, 2024

Acknowledgements

First, I would like to express my deep gratitude to Prof. Dr. Florina M. Ciorba for giving me the opportunity to conduct this Master's thesis and for her constructive and insightful feedback. My special thanks go to Dr. Joana Maria Soares Pereira and Dr. Osman Seckin Simsek for their guidance and support throughout this project. I am very grateful to Michèle Leemann for testing the implementation and providing important user feedback. I would also like to thank Michelle Jessica Bösiger, Michèle Leemann, and Florian Patric Burkhardt for proofreading this thesis. Thanks to the members of the high–performance computing group for the interesting discussions and valuable comments. Finally, I would like to thank all members of the research group of Prof. Dr. Thorsten Schwede at the Biozentrum for the pleasant team atmosphere and an excellent time with many insights into bioinformatics.

Abstract

Genomic context analysis is the study of the genomic neighborhood of a given gene to aid in genome structure and evolution studies or to predict the function of a protein. One tool to support such research is Python–based GCsnap, which generates interactive visualizations of the genomic context of protein–encoding genes. The main limitation of the application is its long execution time, which prevents large–scale studies of entire protein clusters. The poor performance is due to the need to collect data from various online databases via APIs. We present two tools to overcome this limitation: (i) GCsnap2.0 Desktop, designed to run on machines with API-enabled connectivity, and (ii) GCsnap2.0 Cluster, tailored for high-performance computing clusters, where the required data is stored in advance. The evaluation shows that the desktop version outperforms the old implementation, GCsnap1, end–to–end in all cases. However, the performance gain is limited by the speed of the network connection. The cluster variant allows the analysis of entire protein clusters, but there is still room for improvement.

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
- 1.2 Objectives . . . 5
- 1.3 Contribution . . . 6
- 1.4 Outline . . . 6

2 Background **7**
- 2.1 Biological Terminology . . . 7
- 2.2 Parallel Programming Terminology . . . 8
- 2.3 Global Interpreter Lock . . . 9

3 GCsnap **10**
- 3.1 The Workflow . . . 10
- 3.2 Experiments with GCsnap1 . . . 13

4 Related Work **16**
- 4.1 Genomic Context Analysis Tools . . . 16
- 4.2 Parallel Python Tools . . . 16
- 4.3 Parallelization Tools in Practice . . . 19
- 4.4 Summary of Tools . . . 19

5 Parallel Tool Assessment **21**
- 5.1 Tool Discussion . . . 21
- 5.2 Parallel Tools for GCsnap2.0 Desktop . . . 22
- 5.3 ID Mapping with Dask.DataFrame . . . 26
- 5.4 Dask.distributed vs. MPI for Python . . . 27
- 5.5 Summary of the Assessment . . . 29

6 GCsnap2.0 **30**
- 6.1 Results of user survey . . . 30
- 6.2 New Features and Improvements . . . 31
- 6.3 GCsnap2.0 Desktop . . . 33
- 6.4 GCsnap2.0 Cluster . . . 35

7 Evaluation **39**
- 7.1 Consistency of GCsnap2.0 . . . 39
- 7.2 Performance of GCsnap2.0 Desktop . . . 40
- 7.3 Cluster scalability . . . 43

8 Conclusion **45**
- 8.1 Future Work . . . 46

Bibliography **51**

Appendix A: GCsnap User Survey **52**

Appendix B: Additional Plots and Tables **61**
B.1 GCsnap1 Execution Time Analysis 62
B.2 ID Mapping with Dask.DataFrame 65
B.3 Assembly parsing with Dask.distributed and mpi4py.futures 67
B.4 GCsnap2.0 Desktop Performance Analysis 69

List of Acronyms

CDS	Coding Sequence
EMBL	European Molecular Biology Laboratory
EMBL-CDS	EMBL Coding Sequence
EMBL-EBI	EMBL's European Bioinformatics Institute
gc2C	GCsnap2.0 Cluster
gc2D	GCsnap2.0 Desktop
GCA	GenBank (primary) assembly accessions
GCF	RefSeq (NCBI-derived) assembly accessions
GFF	General Feature Format
GIL	Global Interpreter Lock
GO	Gene Ontology
GenBank	NCBI Genetic Sequence Database
HPC	High Performance Computing
I/O	Input/Output
MMseqs	Many–against–Many Sequence Search
miniHPC	HPC cluster of the HPC-Group at the Department of Mathematics and Computer Science
MPI	Message Passing Interface
mpi4py	MPI for Python
NCBI	National Center for Biotechnology Information
PDB	Protein Data Bank
RefSeq	NCBI Reference Sequence Database
sciCORE	Center for Scientific Computing at University of Basel computing cluster
SIB	Swiss Institute of Bioinformatics
SLURM	Simple Linux Utility for Resource Management
TM	Transmembrane
UniParc	UniProt Archive
UniProt	Universal Protein Resource Database
UniProtKB-AC	UniProt Knowledgebase Accession Number

UniProtKB-ID UniProt Knowledgebase Identifier

UniRef UniProt Reference Cluster

WSL Windows Subsystem for Linux

Chapter 1

Introduction

Genomic context analysis refers to the study of the genomic neighborhood of a particular gene to assist in genome structure and evolution studies or to predict the function of a protein [1]. Protein function prediction involves infering the functional associations of newly sequenced genomes by calculating the similarity to proteins for which the function has been identified. A tool for genomic context analysis is GCsnap, a freely available open–source application written in Python [2]. Its purpose is to identify and compare the genomic context of protein–encoding genes by collecting data from various public protein databases, integrating this information, and creating interactive context visualizations [3].

1.1 Motivation

The current implementation of GCsnap, hereafter referred to as GCsnap1, has one major limitation: the performance. Figure 1.1 shows the average execution time for different numbers of input sequences, 1 input sequence is called a target, and CPU cores over 5 repetitions. The key observation is the long end-to-end execution time. It takes on average about 100 seconds to analyze the genomic context of 10 targets. This translates to 10 seconds per target if no parallelism is used, i.e., only 1 CPU core. The reason for this is the number of queries to public databases and online servers. Such operations are limited by the speed of the connection and the efficiency of the database itself to handle the queries, resulting in potentially long latencies.

There is an increasing desire to study the genomic context of large sets of proteins. One use case is the analysis of entire protein clusters from the Protein Universe Atlas [4] consisting of thousands of sequences. Such a task is currently not feasible with GCsnap1 in a reasonable time. One solution is to adapt the tool to run in a distributed environment. High Performance Computing (HPC) clusters provide the necessary resources to perform this kind of large-scale analysis. However, there is a problem with this approach. In most clusters, compute nodes cannot access to the internet, making it impossible to use the online platforms to retrieve data. In order to run GCsnap on such a system, the data must be made available to the compute nodes in advance.

1.2 Objectives

The goal of this thesis is to develop a new version of GCsnap, henceforth called GCsnap2.0, that meets the following requirements:

- **Scalable:** The desired output is produced in a short time, even when analyzing thousands of protein–encoding genes.
- **Portable:** GCsnap2.0 runs on all operating systems, regardless of the computing environment in which it is executed. It should work on both clusters and desktops.
- **Modular:** Currently, GCsnap1 is just a single large Python script. The newly written code is modularized to ensure maintainability and extensibility. This facilitates future customizations and allows components to be used in a package-like fashion.

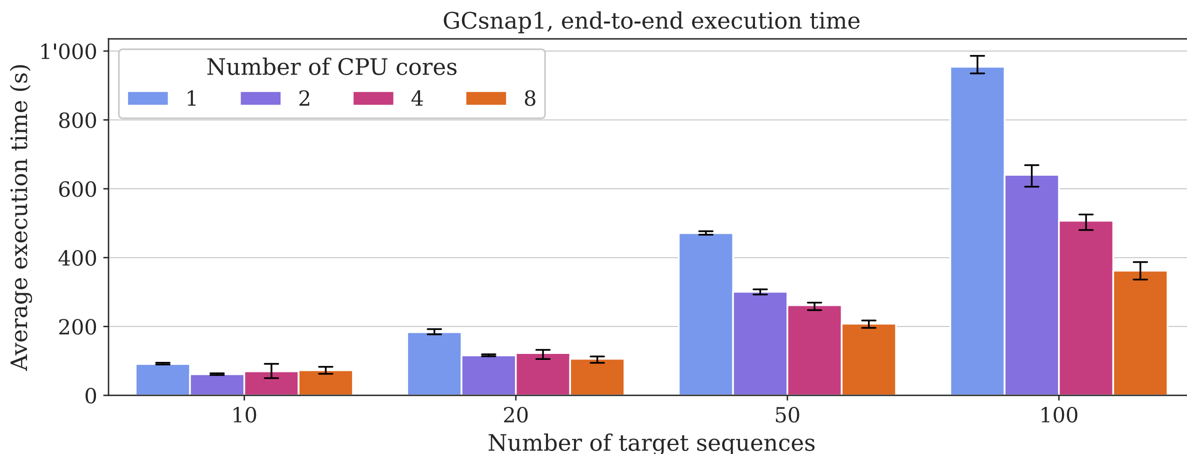


Figure 1.1: Average end-to-end execution time of GCsnap1 over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 CPU with 64 CPU cores.

- **User friendly:** Despite all the new features, the execution of GCsnap2.0 remains simple. If multiple dependencies are added, appropriate software environment handling must be provided.

1.3 Contribution

Given the differences in data access, online versus locally stored data, it is necessary to develop two versions of GCsnap2.0. The main contributions of this thesis are (i) evaluation of suitable parallel Python tools through qualitative and quantitative analysis, (ii) implementation of a performant GCsnap2.0 Desktop (gc2D) for users without access to a computing cluster, and (iii) development of GCsnap2.0 Cluster (gc2C) tailored to run on the Center for Scientific Computing at University of Basel computing cluster (sciCORE).

1.4 Outline

The rest of this thesis is organized as follows. The necessary background information is given in Chapter 2, followed by a detailed description of the GCsnap workflow and the analysis of the GCsnap1 bottlenecks in Chapter 3. Chapter 4 presents an overview of related work before the evaluation of suitable tools is explained in Chapter 5. Chapter 6 describes the implementation, and Chapter 7 the evaluation results. Finally, Chapter 8 concludes the thesis and provides an outlook for future work.

Chapter 2

Background

In this chapter, we provide the essential background knowledge and key terminology that will be used throughout this thesis. The first part is dedicated to biological terms needed to understand the functionality of GCsnap before we introduce parallel programming terminology in general. The final section discusses the Global Interpreter Lock (GIL) and the limitations it imposes on running Python in parallel.

2.1 Biological Terminology

Eukaryotes, Bacteria and Archaea
The three distinct domains of life are eukaryotes, bacteria, and archaea [5, p. 3]. Eukaryotes, such as plants, animals, or yeast cells, have a nucleus that is surrounded by a double membrane inside the cell. In contrast, the nucleoid found in bacteria and archaea is not separated but is simply part of the internal volume of a cell. Bacteria and archaea are unicellular microorganisms formerly grouped as prokaryotes [5, p. 3].

DNA, Genes and Coding Sequences
When we mention a genome, we are referring to all the genetic information encoded in a cell. DNA carries this information. A chromosome is a large DNA molecule that contains many genes. A gene encompasses all the DNA that encodes the primary sequence of a final gene product [5, p. 980]. In this context, the term coding sequence refers to a region in DNA that defines the sequence of amino acids in the protein.

Proteins, Peptides and Amino Acids
A protein is a large molecule composed of 1 or more polypeptides, a chain of amino acids, with each pair of amino acids linked by a peptide bond. When an amino acid is connected to a neighbor, it is referred to as an amino acid residue. All proteins in any organism comprise the same set of 20 amino acids [5, p. 75]. For example, the PhoH–like protein with the identifier PHOL_ECOLI found in the bacterium *Escherichia coli* strain K12 has a sequence of 346 amino acid residues, the first 10 of which are $MNIDTREITL$. Each letter stands for 1 amino acid, for example, M denotes methionine. The complete list of all 20 amino acids and their abbreviations can be found in [5, p. 77]. Throughout this work, we will use the term protein sequence for the actual sequence of amino acid residues.

Protein Structure and Function
The protein sequence is the primary structure of a protein. There are 4 levels of protein structure. Along with the first level, the tertiary structure is mentioned throughout this work. It describes the 3-dimensional folding of the protein [5, p. 97]. These two are closely related, as the protein sequence determines the 3D structure that defines the function of a protein [5, p. 97]. It is important to note that the exact mechanism by which the protein sequence determines the function is not yet fully understood [5, p. 104]. As a result, function prediction from sequence is not always possible. Nevertheless, protein families with a common function can be identified based on the degree of similarity of their sequence of amino acid residues. Regarding their function, two types of protein are of special interest. The first are signal peptides, where the beginning of a gene encodes a signal peptide sequence that marks the protein for export from the cell. The second are Transmembrane (TM) proteins because they span the cell membrane and serve as receptors.

Assembly
Assembly is a functionally relevant complex consisting of two or more proteins [6]. As described in [5, pp. 339-342], one of the approaches to genome sequencing is called shotgun sequencing. Numerous copies of the same DNA are sheared off at random positions into fragments of the desired length. The result is many overlapping segments of the same DNA. The sequence can be traced from fragment to fragment, by piecing together the overlaps, resulting in an assembly of contiguous sequence regions. All the extracted information from genome sequencing is available in so– called assembly files, including the start and end position of a sequence in the genome and its neighboring sequences.

Operons
It is common for bacteria to have operons. In this situation, the genes next to each other on the chromosome form a cluster and are transcribed together [5, pp. 1158-1159]. The process of transcription of clustered genes is initiated by a single promoter. Additionally, regulatory sequences serve as binding sites for proteins to either activate or repress transcription; the latter is called an operator. Together, the promoter, genes, and regulatory sequences form the operon. In general, the encoded products of the genes in an operon have interdependent functions. Bacterial genomes usually contain a few highly conserved operons [7], meaning that their sequences have a high degree of similarity.

2.2 Parallel Programming Terminology

Parallelism vs. concurrency
Parallelism is the execution of multiple tasks at the same time. This is different from concurrency, where multiple tasks are interleaved, but only one is running at a time. As a simple example, the former is a situation where we have two queues and two machines, while the latter is a situation where we have two queues but only one machine. In this case, the machine can only serve one queue at a time. Progress is achieved by switching between the concurrent queues. With two machines, however, both queues are processed in parallel. In this sense, parallelism is a property of the hardware, whereas concurrency is a property of the software.

Computing node architecture
The architecture on which a script is executed is crucial for parallelism. This is especially important when considering nodes in a cluster, where multiple nodes are connected by a fast network. The schematic of an example node is shown in Figure 2.1. The node contains two sockets, each with 1 CPU containing 4 cores, where each core has equal access to the local memory. Together, the two CPUs form a distributed memory situation because they do not have direct access to each other's memory. There needs to be some means of exchanging information. In this case, this means both inter–socket communication and internode transfer. This is generally slower than communication between cores on the same socket, due to the higher latency and lower bandwidth associated with inter-socket communication. It is important to note that different computer architectures require different programming paradigms.

Processes vs. threads
A process is a running program with its own allocated memory. Threads within the same process share memory. Threads are the basic unit of CPU usage, so each process contains one or more running threads, with the latter called a multithreaded process. Because all threads of the same process share the same address space, interaction between them is faster, but more prone to concurrency problems. Concurrent use of shared resources can lead to race conditions. A situation in which two concurrent threads access the same variable in memory, leading to unintended behavior if not properly handled. On the other hand, context switching between threads within the same process is faster than between processes due to the shared memory. Context switching is when the operating system replaces threads in a waiting state on the CPU with threads that are running.

Data vs. task parallelism
Data parallelism involves performing the same operation on subsets of the same data. For example, consider summing an array, where one thread sums the first half and a second thread handles the second half of the array. This type of parallelism focuses on using data across multiple CPU cores, allowing each core to perform the same computation on different chunks of data. Data parallelism often requires

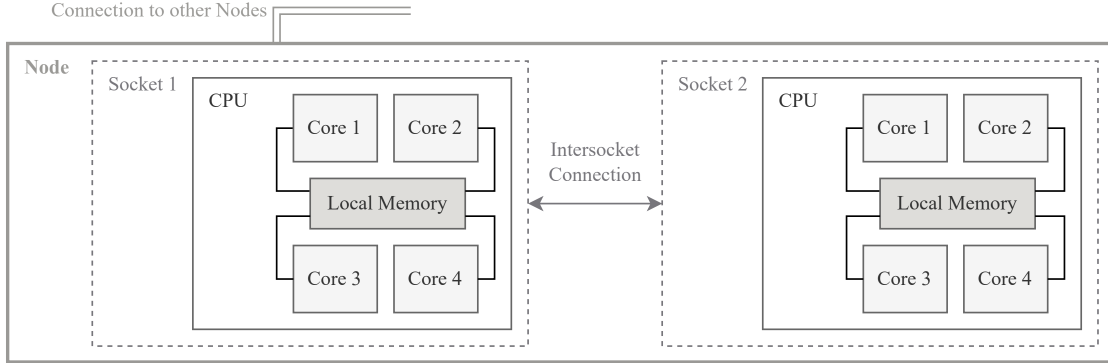


Figure 2.1: Schematic of a compute node with two sockets, each equipped with a 4-core CPU, where each core has direct access to its local memory. Access to the memory of the other CPU requires communication through the intersocket connection. Multiple nodes are connected by a high–speed network.

synchronous execution, meaning all threads must complete before the results can be combined. Data parallelism also works with processes, except that data must be distributed in advance because processes do not share memory.

Task parallelism involves dividing a program into separate tasks that can be executed simultaneously. Unlike data parallelism, where the same operation is applied to different data, task parallelism allows each thread or processor to perform a different operation. When there is no dependency between tasks, execution can be asynchronous. Task parallelism is more general and flexible, since it can handle a wider variety of tasks with different computational requirements.

2.3 Global Interpreter Lock

To avoid potential concurrency problems, the default CPython interpreter has the GIL [8]. This mechanism ensures that only one Python thread executes bytecode at a time, even on multicore machines, thereby guaranteeing security against concurrent access by design. The threading module is the basic tool for concurrent execution in Python [9]. Threads are created by passing a function and arguments to the class constructor. Each thread has a reference to these arguments and its own copies of local variables. The module does not achieve parallelism but only supports concurrency. However, threading is useful for overlapping concurrent Input/Output (I/O) operations to reduce overall wait time. To run Python in parallel despite the GIL, we must work with processes provided by modules or third-party tools, as discribed in Section 4.2.

When talking about the GIL, it is important to mention the ongoing effort to make it optional. The accepted Python Enhancement Proposals (PEP) 703 of 2023 details the necessary changes to the CPython internals to support running without the GIL [10]. Given the far–reaching consequences of such a change, the idea is to include a build configuration flag to build CPython to run without the GIL, but with GIL remaining the default. In fact, this feature is available in the first release candidate of Python 3.13 from August 2024 as an experimental free-threaded build mode [11, 12]. Due to the experimental nature and the late release for this work, we did not consider it as an option for our implementation.

Chapter 3

GCsnap

The first part of this chapter covers the general functionality of GCsnap, where an explanation of the workflow and the details of the data used is given. The second part is dedicated to GCsnap1, talking about the current implementation and presenting the results of the experiments we conducted to identify the causes of poor performance.

3.1 The Workflow

The purpose of GCsnap is to provide a comparison of the genomic context of genes that encode proteins [2]. It works for both prokaryotes and eukaryotes. The tool handles various input formats, collects information from different protein databases, searches for similar proteins, identifies clusters, and combines all this information into interactive plots. The organizations providing and managing the vast amount of information include the National Center for Biotechnology Information (NCBI), the Universal Protein Resource Database (UniProt), the European Molecular Biology Laboratory (EMBL), and the Swiss Institute of Bioinformatics (SIB). A workflow visualization is shown in Figure 3.1. The various tasks performed by GCsnap can be divided into three blocks: collect, find families, and annotate. Those which will be discussed in the following, along with input and output. The focus here is on the tools and data used. More details about GCsnap can be found in [3]. An overview of all data is presented in Table 3.1.

Input
GCsnap takes sequence identifiers, called targets, of protein-coding genes as input. It supports many identifier standards, which are listed below. For illustrative purposes, we added the identifier for each standard of a PhoH–like protein sequence found in the bacterium *Escherichia Coli* strain K12:

- NCBI Reference Sequence Database (RefSeq): NP_415193.2
- UniProt Knowledgebase Identifier (UniProtKB-ID): PHOL_ECOLI
- UniProt Knowledgebase Accession Number (UniProtKB-AC): P0A9K3
- GeneID: 86863170
- UniProt Reference Cluster (UniRef): UniRef100_P0A9K5, UniRef90_P0A9K5, UniRef50_P0A9K5. Those are actually clusters, where the number represents the percentage of equality among cluster members.
- UniProt Archive (UniParc): UPI0000163983
- EMBL Coding Sequence (EMBL-CDS): AAB40862.1
- Ensembl: No identifier for Escherichia Coli, but an example is ENSG00000139618.

The sequence identifiers can be provided either as (i) a simple list, (ii) a text file with each identifier on a separate line, (iii) a FASTA file, or (iv) a sequence cluster file in CLANS format. While FASTA is a text-based format for representing many protein sequences in one file, the CLANS format is specialized for working with the CLANS (CLuster ANalysis of Sequences) bioinformatics toolkit developed and hosted at the Max Planck Institute for Biology, Tübingen [14, 15].

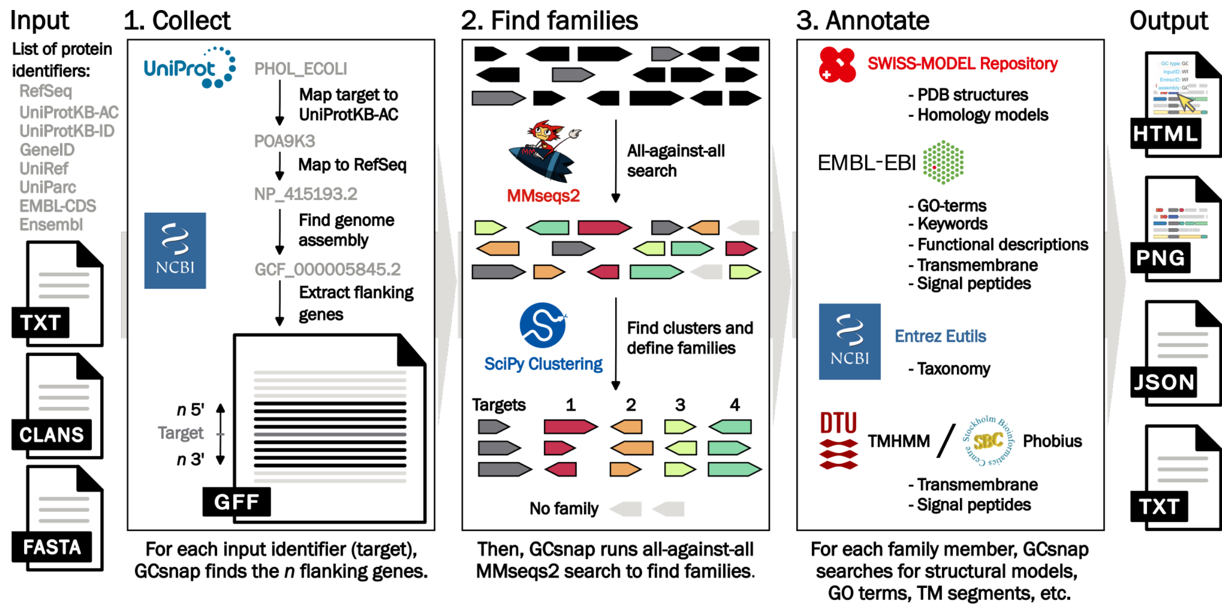


Figure 3.1: Schematic workflow of GCsnap with its three main task blocks. Starting from a list or text file with sequence identifiers from different protein ID standards, GCsnap finds the assembly and extracts the n neighboring genes around the target, called flanking genes, in block 1. Block 2 is responsible for family discovery. All collected genes are clustered based on an all–against–all similarity search of the encoded proteins with MMseqs2 [13] before being grouped into families. In block 3, GCsnap annotates families with structures and functions and family members with sequence features and taxonomy information from various sequence repositories and online tools. The collected information is stored in various text files and visually presented through static and interactive images. This schematic adapts the original in [3, Figure 1].

Block Collect

The first task of GCsnap is to collect all information about the neighboring genes of a target from the genome assemblies. This process involves the following steps:

- Each input protein sequence identifier is first mapped to UniProtKB-AC and then to RefSeq or EMBL-CDS using the UniProt API. This is necessary because the genomic assembly file can only be found with these 2 ID standards.

- The protein sequences adjacent to the target in the genome are extracted from the assembly files. Those are General Feature Format (GFF) files, a format specialized for describing genes but readable like any other text file. They are available from the NCBI Genetic Sequence Database (GenBank) or the NCBI Reference Sequence Database (RefSeq). Using the mapped ID, GCsnap requests the assembly accession from the NCBI Eutils API and retrieves the URL pointing to the assembly file from the summary tables provided by the NCBI (see Table 3.1). The difference between GenBank (primary) assembly accessions (GCA) and RefSeq (NCBI-derived) assembly accessions (GCF) is that GCA contains records submitted and owned by researchers or sequencing centers. In contrast, GCF records are maintained by NCBI staff and always include annotations [16]. In cases where both are found, GCsnap favors RefSeq.

- Using the retrieved URL, GCsnap downloads the assembly file and extracts the neighboring genes around the target, called flanking genes, within the same sequence region. By default, 4 such flanking genes are collected on the downstream (3', pronounced "three–prime") and upstream (5') sides. If there are fewer flanking genes on either side, it represents a partial genomic context block. Users can specify whether or not to keep these for further processing.

- The coding sequences of the target and its flanking genes are obtained from NCBI using the Eutils API. The Coding Sequence (CDS) is the protein's actual sequence of amino acids. Additionally, the API returns the taxonomy ID and the name of the species of the target.

Block Find Families
This block aims to identify protein families among all targets and their flanking genes collected in the previous block. This involves two steps: First, an all–against–all search on all CDS to find sequence similarities, and second, finding clusters within the similarity matrix using the *cluster.hierarchy* tool from SciPy [17]. Each cluster represents a family. GCsnap1 provides two methods for calculating similarity scores between the CDS: Basic Local Alignment Search (BLAST [18]) or Many–against–Many Sequence Search (MMseqs). The former involves building a protein BLAST database (BLASTp) against which each protein sequence is searched during position-specific iterative BLAST (PSI–BLAST). Compared to PSI–BLAST, MMseqs is more than 400 times faster with the same sensitivity [13]. For this reason, GCsnap2.0 will no longer include BLAST and will only support MMseqs.

Block Annotate
The purpose of this task block in GCsnap is to annotate the families and the members of each family. Additional family features and sequences information includes:

- The URL pointing to the Protein Data Bank (PDB) 3D structure is retrieved from the SWISS–MODEL repositories or, if not found, through the AlphaFold API.
- Gene Ontology (GO) terms, i.e., information on the functions of genes, are queried with the EMBL's European Bioinformatics Institute (EMBL-EBI) API. Furthermore, the functional description, TM–topology, and additional keywords are retrieved.
- The taxonomy of each protein sequence is requested from NCBI Eutils with the taxonomy ID collected in block 1. GCsnap uses the taxonomy to build the phylogenic tree shown in the interactive output.

The final step is to identify the presence of TM segments and signal peptides. GCsnap either requests the data from EMBL-EBI or allows the use of a software tool. The user can choose Phobius [19] or

Table 3.1: Details about the data collected by GCsnap in the first task block.

Data	Source	URL	Accessed through
Mapping between ID standards	UniProt	https://www.uniprot.org/help/programmatic_access	API
Assembly accession, coding sequences, taxonomy	NCBI Eutils	https://eutils.ncbi.nlm.nih.gov/	API
Assembly summary tables	GenBank	https://ftp.ncbi.nlm.nih.gov/genomes/genbank/assembly_summary_genbank.txt	TXT file
	RefSeq	https://ftp.ncbi.nlm.nih.gov/genomes/refseq/assembly_summary_refseq.txt	TXT file
Assembly files	GenBank	https://ftp.ncbi.nlm.nih.gov/genomes/all/GCA/	GFF files
	RefSeq	https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/	GFF files
3D protein structure URL	SWISS-MODEL Repository	https://swissmodel.expasy.org/repository/	API
	AlphaFold	https://alphafold.ebi.ac.uk/	API
GO-terms, keywords, functional description, TM-topology	EMBL-EBI	https://www.ebi.ac.uk/proteins/api/	API

TMHMM 2.0 [20]. Although it is noted that the current version used by GCsnap is outdated, the new deep learning based DeepTMHMM [21] was not considered for our work. GCsnap runs the selected tool with a FASTA file containing all flanking genes as input when installed. Alternatively, the tools can be run online, and their output can be used as an input to GCsnap.

Output
The most informative output GCsnap creates is the HTML plot, a graphical interactive summary of the results that can be explored further by clicking on genes of interest. An explained illustration can be found in [3, Figure 2]. Other output produced includes plots, summary files in JSON format containing all collected information about the genomic context, and TXT files with additional details.

3.2 Experiments with GCsnap1

The goal of this section is to evaluate how well GCsnap1 handles many input sequences and to identify the causes of the long execution time despite exploiting concurrency by running multiple threads on different cores through Multiprocessing's ThreadPool construct (see Section 4.2 for details). Identifying the bottlenecks is crucial to know where to focus when implementing GCsnap2.0. To achieve this, we conducted a series of experiments, measuring the end-to-end execution time of the three task blocks and of individual steps with the version available in [2, Branch: timing]. They were executed on the HPC cluster of the HPC-Group at the Department of Mathematics and Computer Science (miniHPC). The design of the factorial experiments is shown in Table 3.2. In these experiments, GCsnap1 was run without specifying the personal NCBI API key. Therefore, we were limited to 3 requests per second when querying the Eutils API [22]. While the higher limit of 10 requests per second may yield better performance, we wanted to analyze the bad–case scenario.

Scaling analysis
The empirical results with up to 2'000 targets for different numbers of CPU cores are shown in Figure 3.2. Although the end-to-end execution time for this argument setting is very long, GCsnap1 does scale. The execution time decreases almost linearly with the number of cores used, strongly suggesting that the parallelization technique used works as expected. More results for different numbers of target sequences and CPU cores can be found in Appendix B.1.

However, the analysis with numerous targets and many CPU cores revealed another problem with GCsnap1. When examining the individual results for each repetition of a specific argument combination, we noticed that a total of 67 experiments did not finish. This occurred across 26 different combinations

Table 3.2: Design of factorial experiments to analyze the performance of GCsnap1.

Factor	Value	Properties
Application	GCsnap1	Number of targets: 10, 20, 50, 100, 200, 500, 1'000, 2'000 Number of CPU cores: 1, 2, 4, 8, 16, 32, 64 Runtime arguments for GCsnap1: -annotate_TM True -all-against-all_methods mmseqs -ncbi_api_key None
Metrics	Program performance	Execution time (seconds) end-to-end, of the three task blocks, individual steps
Computing system	1 miniHPC node	2 AMD EPYC 7742, 2.25 GHz; each with 64 cores and 256 MB L3 cache; 1'500 GB RAM; API-enabled connectivity
Validity	Repetitions	5

Figure 3.2: Average end-to-end execution time of GCsnap1 over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 CPU with 64 CPU cores. Not all repetitions finished.

of CPU cores and number of targets. For example, with 2'000 targets and 16 CPU cores, 3 out of 5 repetitions failed. All failed attempts are listed in Appendix B.1.

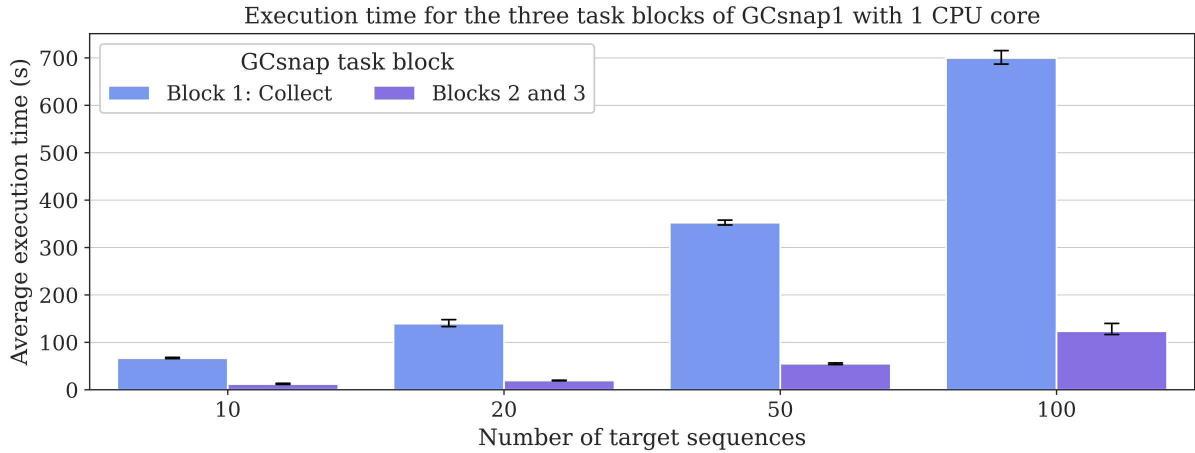
There is no apparent pattern, and the problem appears to occur randomly. Furthermore, the reasons for failed experiments are unknown because the version of GCsnap1 measuring execution time catches most errors without proper handling or reporting. By examining the output of each experiment, we determined at what step GCsnap1 failed. 12 failed during the collection of genomic context, 10 failed during the collection of taxonomic information, and the majority of the 45 experiments failed during the functional annotation step. All 3 steps have in common that they rely on requesting data from an online API. The fact that failures occur more frequently when many CPU cores are used suggests that the problem is related to API limitations. When a request is blocked, a certain number of retries are attempted, and if unsuccessful, the request is eventually aborted.

In summary, this indicates that GCsnap1 was not designed to handle large workloads. On the one hand, the very long end-to-end execution requires more computing resources. On the other hand, using more CPU cores leads to more failures.

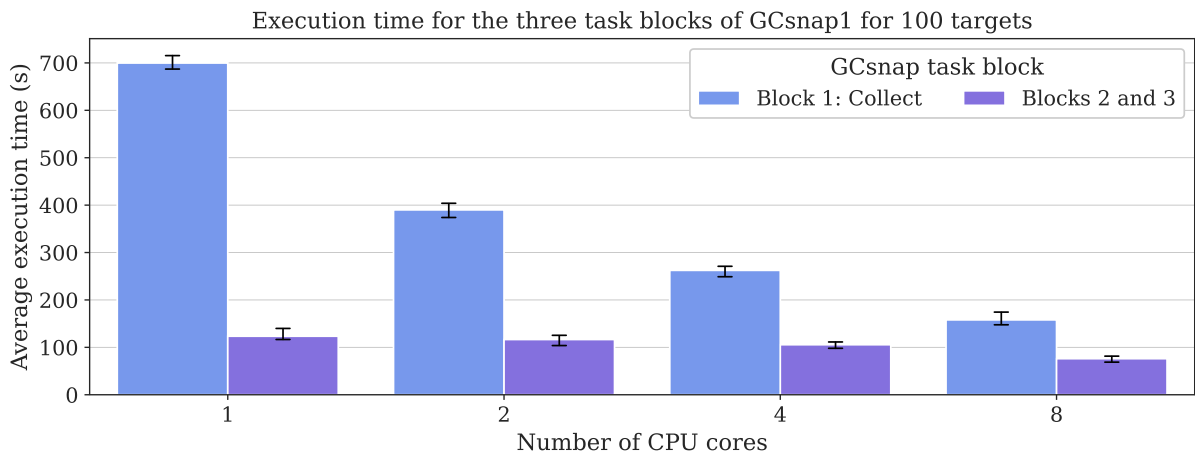
Identifying bottlenecks

To identify the steps that cause the poor performance of GCsnap1, we measured the execution time of the three task blocks and individually for each step. Figure 3.3 shows the time to complete the three task blocks of GCsnap1. Two noteworthy observations: First, the collection of genomic context information is primarily responsible for the poor performance of GCsnap1. Regardless of the number of targets, this task block takes the most time (Figure 3.3a). Second, only the first block benefits from using more resources. As shown in Figure 3.3b, the collection of genomic context scales with the number of CPU cores, while blocks 2 and 3 together show little improvement.

Based on these findings, it is evident that the focus must be on the first block of GCsnap. Figure 3.4 shows the execution time of the individual steps within block 1: (i) finding the NCBI assemblies, (ii) downloading and extracting the assemblies, and (iii) adding the actual sequences to the flanking genes. IT is clear that the first and third steps cause GCsnap1's inefficiency. As mentioned in Chapter 1, GCsnap1 takes an average of about 10 seconds per sequence. The height of the bars in Figure 3.4 indicates that finding the assemblies and adding the flanking sequences take 2 and 4 seconds per target, respectively. On average, these steps account for 60 percent of the GCsnap1 execution time per target, making them the clear bottlenecks.



(a) The average execution time with various targets with 1 CPU core.



(b) The average execution time with various CPU cores and 100 input targets.

Figure 3.3: Average execution time of GCsnap1 for the three task blocks over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 CPU with 64 CPU cores. Not all repetitions finished.

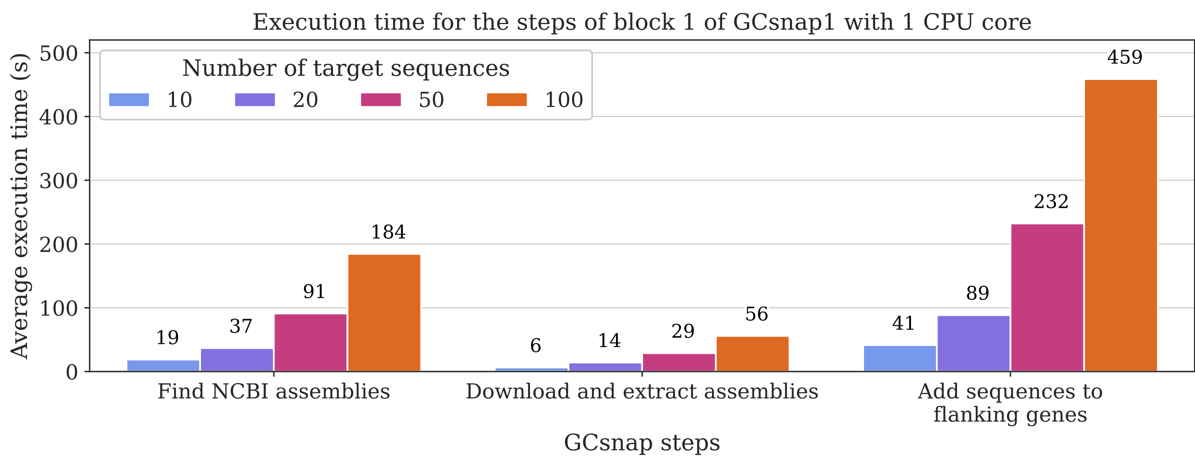


Figure 3.4: Average execution time of the steps performed in task block 1 over 5 repetitions with different numbers of CPU cores and input targets. Experiments were conducted on an AMD EPYC 7742 CPU with 64 CPU cores. Not all repetitions finished.

Chapter 4

Related Work

In this chapter, we review studies and tools relevant to our work. First, we discuss existing strategies for genomic context analysis. Second, we introduce third-party tools used to execute Python code in parallel in various ways. Third, we present studies that rely on these tools, and finally, we summarize the tools reviewed and discuss their interrelationships.

4.1 Genomic Context Analysis Tools

Several tools similar to GCsnap have been developed in recent years. One example is FlaGs [23], which allows users to create visualizations for genomic context analysis. The tool is named after flanking genes. The supported input identifiers are limited to RefSeq, as the tool exclusively uses GCF assemblies. This differs from GCsnap, which supports many ID standards and uses assemblies from both RefSeq and GenBank. FlaGs uses Jackhammer, a method based on Hidden Markov Models, to find clusters among the flanking genes. Unlike GCsnap, the generated visualizations are not interactive. However, FlaGs offers both a web–based version and the option to run it locally. A modification of FlaGs called NetFlax was used in [24] to analyze proteinaceous toxin–antitoxin (TA) systems.

Another representative of web–based tools is GeCoViz [25], which creates interactive visualizations of the genomic context of prokaryotes. GCsnap can handle sequences from both prokaryotes and eukaryotes. GeCoViz relies on precomputed information stored in a MongoDB database to provide fast search results. Another web application limited to prokaryotes is the Microbial Genomic Context Viewer (MGcV) [26]. Similar to GeCoViz, there is a MySQL database in the backend which is updated on a weekly basis. The difference lies in the stored data. While GeCoViz stores the pre-calculated information, MGcV stores the data needed for the calculation to ensure fast access.

Another tool, the Genomic Context Viewer (GCV) version 2 [27], takes a slightly different approach. This web–based application is designed for visualizing microsyntenies, small genomic regions derived from a common ancestor. GCV analyzes relationships in real–time through on–demand computation across federated data sources. It is also possible to integrate GCV into existing web applications.

4.2 Parallel Python Tools

Many existing Python packages for scientific computing are designed to overcome the limitations of the GIL. The most prominent of these is NumPy [28]. For example, when Python computes the sum of two NumPy arrays, such as `C = A + B`, the GIL is freed allowing another thread to run. The same goes for Pandas [29], a Python library that provides table-like data structures called dataframes, where certain functions like `groupby` release the GIL. Another way to bypass the GIL is to use packages with a particular compiler, for example Numba [30], which is designed for parallel scientific computing. It uses a just–in–time (JIT) compiler, which reads the bytecode of a function decorated with `@jit` and compiles it to produce a machine code version. Each time the function is called, the compiled version is used without the need to hold the GIL.

There are many other solutions for exploiting parallelism in Python. Some examples focus on parallelizing Python on a single machine, but several projects also attempt to orchestrate execution on a

heterogeneous node of CPUs and GPUs, or in a distributed environment. Representatives of each group are presented below.

Multiprocessing

The Multiprocessing package, a part of the standard Python library, overcomes the GIL using processes instead of threads, as explained in [31]. The main purpose is to run different processes in parallel on a multicore machine. In addition, the package provides the `Pool` object to execute a function (i.e., a callable) with different input values in parallel, making it an ideal tool for enabling data parallelism. Processes from `Pool` can be executed synchronously or asynchronously. Furthermore, the library supports parallelism with threads through the `ThreadPool` construct.

Concurrent.futures

As described in [32], the concurrent.futures module provides an interface for asynchronous execution of callables. It is also part of the standard Python library. It uses futures, where an object is returned immediately when the function is called, rather than commonly when the function terminates. The module supports the use of either threads with the `ThreadPoolExecutor`, or processes, using the `ProcessPoolExecutor`. The first is built on top of the Threading module and has an implicit barrier, as all threads entering the pool are joined before the interpreter can exit. The second uses the Multiprocessing package under the hood [32, par. ProcessPoolExecutor]. It is important to note the fundamental difference: while threads are not executed in parallel due to the GIL, processes created with the `ProcessPoolExecutor` are.

MPI for Python

As mentioned in Section 2.3, processes are not constrained by the GIL. The challenge lies in inter–process communication. The Message Passing Interface (MPI) is one of the standard application program interfaces that allows information to be exchanged between CPUs in distributed systems. The standard [33] defines routines provided by libraries for use in traditional HPC languages such as C/C++ and FORTRAN. When running a program with MPI, all processes form a group called the world communicator. All members, usually called ranks, are connected to each other. There are two major communication primitives: Point-to-Point and Collective. The former involves only two processes, while the latter involves all processes.

MPI for Python (mpi4py) is a package that provides bindings to access the MPI routines directly from a Python script. Development of this package started in 2004 and is continuously adapted to new MPI standards [34]. The package allows Python objects such as NumPy arrays to be passed in various ways, including non–blocking versions of both communication primitives and buffered messages. It also supports the creation of MPI groups, subsets of processes that form a group communicator within the world communicator.

One drawback to the ease of use of mpi4py is the need to write your code in MPI style. The programmer has to manage the master–worker pattern, where rank 0 is usually the master and all other ranks are the workers. To address the community's need for an easier way to deploy computation on an HPC cluster, the mpi4py.futures module was introduced in [35]. It is based on concurrent.futures and can be used in the same way through the `MPIPoolExecutor`. Details about the command and how to use it can be found in [36].

Dask

Dask is a flexible parallel computing library first introduced in [37]. It encodes task graphs using Python data structures, namely dictionaries, tuples, and callables. In these graphs, the nodes represent individual tasks and the edges represent data dependencies between tasks [38]. The library's scheduler executes the graph simultaneously, while respecting the dependencies. Parallelism is achieved using Dask's *Futures*, derived from the concurrent.futures module. Parts of the distributed computation are built on top of mpi4py, particularly the *Dask-MPI* module.

In addition, the library provides special tools for distributing large data structures to overcome memory limitations. A *Dask.Array* is composed of smaller pieces of NumPy like arrays. Since NumPy releases the GIL, operations can be performed in parallel. Similarly, a *Dask.DataFrame* is a collection of Pandas data frames.

Dask offers different usage methods depending on the machine it is running on. The simple case for a local machine or single node on a computing cluster is `LocalCluster`, which sets the specified number of CPU cores. The `Client` connects to this local cluster, to create and execute the task graph. `Client` also

simplifies the process by allowing you to specify the desired resource directly when creating an instance of `Client`. The ability of Dask to run on HPC clusters is much more relevant to this thesis is. The *Dask.Distributed* module provides an API for interacting with cluster management systems such as the Simple Linux Utility for Resource Management (SLURM) [39]. Using `SLURMCluster`, it is possible to request resources on an HPC system directly from Python. Under the hood, this simply involves sending SLURM jobs to the cluster requesting additional compute nodes. Once SLURM assigns these nodes to Dask, the worker nodes execute the task graph managed by the Dask scheduler running in the main Python thread.

PyCOMPSs

PyCOMPSs [40] is a parallel programming framework for Python that facilitates the development of workflows for distributed systems. It is built on top of the COMPSs Java runtime system, which provides the language bindings for Python applications. In order to execute callables as asynchronous parallel tasks, functions in a sequential Python script are annotated with the `@task` decorator. PyCOMPSs creates a task for each decorator invoked.

The runtime system exploits the inherent concurrency of the script and assigns tasks to available resources while enforcing the detected dependency graph. The return values of callables are represented as futures to ensure task execution is asynchronous. Available resources are specified in configuration files. The scheduling policy is location–aware, meaning a score is calculated for each resource, based on the number of input parameters already present on that resource. The scheduler assigns the task to the resource with the highest score, ensuring that as little data as possible is transferred.

Parsl

Similar to PyCOMPSs, Parsl decomposes data dependencies into a dynamic task graph [41]. The main difference to PyCOMPSs is the absence of a runtime system from another language, as Parsl is completely implemented in Python. The core functionality is the *App* decorator `@python_app` and the *future* object. Parsl registers an asynchronous task when invoking a decorated function and immediately returns a *future*. *DataFlowKernel* manages the construction of task graphs and their order of execution. The nodes represent the *Apps* to be executed, and the edges are derived by passing *futures* between *Apps*.

To ensure portability across computing systems, Parsl includes the *Provider* interface. This abstraction handles three actions: Submitting a job, getting the status of a job, and canceling a running job. The *Provider* supports interaction with various cluster management tools, including SLURM, AWS, Google Cloud, and Kubernetes. The actual execution is handled by Parsl's *Executor*, which provides a collection for common execution patterns, such as high throughput or extreme scale problems. Under the hood, it is an extension of the concurrent.futures module along with mpi4py to manage distributed execution.

Parla

It is a tasking system for Python introduced in [42]. Parla's runtime module implementation allows the orchestration of kernel and library functions within a single process. To overcome the limitations imposed by the GIL, Parla interoperates with NumPy and Numba. Unlike Dask, PyCOMPS, and Parsl, which are designed for workflow management on distributed systems, Parla focuses on managing heterogeneous resources on a single node, such as nodes containing CPUs and accelerators like CUDA–enabled GPUs.

Tasks in Parla are arbitrary blocks of Python code. Using the `@spawn` decorator, these blocks are executed asynchronously. Unlike the above tools, it explicitly uses blocks instead of functions to decouple functional abstraction from parallelism, making it easier to add parallel code blocks around sequential code without restructuring. There are no implicit barriers where a task waits for others to complete. Tasks are assigned to worker threads at runtime, and placement on a device can be specified via the decorator. The same applies to task names. The programmer defines task dependencies using the `dependencies=[task_name]` option.

Flux

Traditional HPC management software such as SLURM has limitations in handling complex workflows consisting of many heterogeneous small tasks [43]. The Flux Framework [44] is a collection of C and C++ projects, libraries, and tools that provide resource managers and schedulers for large HPC centers. It features a modular architecture and a hierarchical scheduling model to support parallelism for high throughput. This model allows each Flux instance to spawn child instances of arbitrary depth and

width. Users can choose scheduling policies within an instance through an API. Recently, a module was developed to support Python bindings to a running Flux instance [45].

ExaWorks
One approach to combining several of the tools described above is ExaWorks, presented in [46]. The goal was to create a workflow software development toolkit (SDK) to enable the composition and interoperation of existing workflow management tools. These technologies form the first pillar of the ExaWorks technical approach. The second pillar is the Portable Submission Interface for Jobs (PSI/J), an abstraction layer that serves as an API for interacting with various HPC workload schedulers to create portable workflows [47]. The third pillar is the SDK itself, which supports the easy installation of the workflow systems [48]. Of interest for our work are the prototyped integration architectures. For example, Parsl can be combined with Flux to overcome the limitations of Parsl's executor to schedule tasks based on resource requirements.

4.3 Parallelization Tools in Practice

The tools presented above have been used in various studies and projects. An example using concurrent.futures is described in [49]. The idea was to simulate hyperspectral data from existing multispectral and hyperspectral data. The process involves computing Chebyshev distances between many pixels of satellite imagery. By using standard Python modules, the execution time was significantly reduced. However, the code was not designed to work with distributed computing resources.

An example of distributed computation using mpi4py is presented in [50]. The tool generates US East and Gulf Coasts meshes to simulate compound flooding. The approach extended the one-dimensional thalweg, the line of lowest elevation within a valley, from digital elevation models to a 2D river representation. These were then used to support mesh generation. Parallelization was necessary to enable continental scale computation.

Another application using mpi4py and NumPy can be found in [51]. The goal was to simulate hybrid particle–field molecular dynamics (hPF–MD), a coarse-grained view of MD in which a group of atoms is treated as a single entity. Additionally, instead of particle–particle interactions, hPF models are based on particles interacting with an external density. Massive parallel I/O was enabled by using the specialized HDF5 binary data format. The corresponding Python library, h5py, allows easy manipulation of huge amounts of NumPy data [52].

In [53], two use cases of parallel execution for social sciences were presented: social network (SN) simulation and kernel polynomial method (KPM). The former is an example of task parallelism, where each propagation of a tweet through the network is a task, while the latter involves data parallelism in computing the eigenvalues of the adjacency matrix of the graph. Both applications were implemented using mpi4py and Python scientific computing libraries such as NumPy. The authors also used Numba to speed up random number generation and list sampling. The paper also describes strategies to reduce the execution time further. For the SN simulation, this included changing the order of task execution to achieve better load balancing. In the case of KPM, improved data distribution was implemented to match the available hardware architecture, specifically, the number of MPI groups created is equal to the number of NUMA domains available on a given computing platform.

A significant project is MQCAS [54], the Quantum Chemistry Archive (QCArchive) of the Molecular Sciences Software Institute (MolSSI). It is a central server for collecting and hosting QC data and making it available to the molecular science community. Of interest to our work is the Python-based open–source QCArchive infrastructure. The important component is *QCFractal*, a server that provides a central task queue where tasks are evaluated by a single or multiple so–called managers. These managers interact with physical resources ranging from workstations to supercomputers. The share of resources used by each manager is defined by a configuration. To achieve high throughput distributed computing, the managers rely on systems such as Dask and Parsl, which interface with queuing systems such as SLURM. To store computations, *QCFractal* uses a PostgresSQL database.

4.4 Summary of Tools

An overview of the tools presented can be found in Figure 4.1. It illustrates the relationship between the different tools and emphasizes the hierarchical structure. The innermost circle contains standard Python

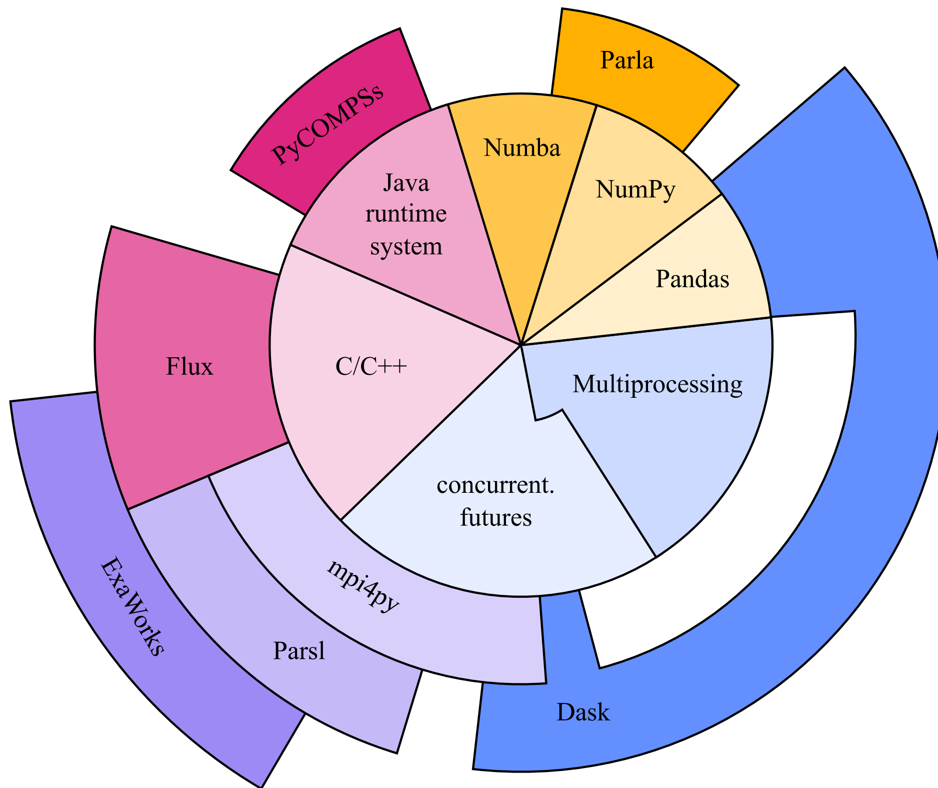


Figure 4.1: Visualization of all presented tools and their connection. The colors in the center circle indicate the type of the tool. Blue for standard Python library modules, yellow for extension packages and red for other programming languages. Reading examples: ExaWorks relies on Flux and Parsl. Concurrent.futures relies on Multiprocessing.

modules, extension packages, and other programming languages, all of which are used to overcome the GIL. The outer rings represent solutions that utilize the adjacent inner tools. For example, ExaWorks relies on Flux and Parsl, with the latter using mpi4py which is based on concurrent.futures, which in turn partly relies on Multiprocessing. The further a ring is from the center, the more functionality the tool provides. Mighty tools offer advantages such as including schedulers or creating task graphs for more efficient execution. However, some of these can also add complexity to deployment and execution. Because they are ongoing projects, these tools are more prone to changes than standard modules. In contrast, the inner circle solutions are easier to implement but have limited capabilities. It is important to balance this tradeoff for each future design decision. The next chapter discusses the suitability of each tool for our work.

Chapter 5

Parallel Tool Assessment

This chapter discussed the suitability of the parallel Python tools presented in Section 4.2 for our work. As mentioned in Section 1.3, we are implementing two applications, GCsnap2.0 Desktop (gc2D) intended to run on any laptop, and GCsnap2.0 Cluster (gc2C) tailored for HPC systems. Each has specific requirements and necessitates different tools. In addition to a purely qualitative discussion, we conducted experiments to evaluate the tools under consideration for quantitative analysis.

5.1 Tool Discussion

An important criterion in evaluating tools is to keep things as simple as possible. Regarding Figure 4.1, we prefer solutions from circles closer to the center. Here, we explain our reasoning for considering some tools and omitting others. A summary is shown in Table 5.1, where we indicate whether each tool is being considered for a particular application and why.

Table 5.1: A summary table assessing each tool, including a column indicating if and for which of the applications it is being considered. Other programming languages are not listed because they were never considered.

Tool	Considered	Reasons
Multiprocessing	gc2D	Standard way to have process and thread parallelism
concurrent.futures	gc2D	Standard way to have process and thread parallelism
NumPy	gc2D and gc2C	Arrays as basic data structures
Pandas	gc2D and gc2C	Dataframes as basic data structures
Numba	No	Computationally intensive part solved with MMseqs
Parla	No	No heterogeneous computing resources to manage
PyCOMPSs	No	Difficult to deploy due to required infrastructure
Flux	No	Difficult to deploy due to required infrastructure
Dask	gc2D and gc2C	Process and thread pool to enable parallelism, handle large files, distributed computing
mpi4py	gc2C	Standard for distributed computing
Parsl	No	Similar capabilities as Dask, but missing the feature to handle large files
ExaWorks	No	Difficult to deploy due to required infrastructure

Considered tools

The primary objective is to parallelize Python by using processes to overcome the limitations imposed by the GIL (see Section 2.3). Since GCsnap heavily uses URL requests, we are also considering using threads. Suitable tools include the pool executors from Multiprocessing, concurrent.futures, and Dask's `Client` command. All three support both processes and threads. While they are sufficient for gc2D, they are inadequate for implementing gc2C, as we need means for inter–process communication.

The straightforward approach for gc2C would be to use mpi4py or *Dask.Distributed*. Since Dask is already a possible solution for gc2D, we prefer Dask instead of mpi4py. Additionally, Dask could help solve another foreseeable problem. In the case of gc2C, we need all the data available on the cluster. Some files are large and may not fit into memory. In particular, the table containing all the mappings between ID standards is 45 GB in size. See Section 6.4 for details. *Dask.DataFrame* is designed to work with such large files.

When implementing gc2C and gc2D, we use Pandas data frames as an easy-to-use data structure for storing data in tabular format. Similarly, NumPy arrays are used for efficient data handling.

Not considered tools

Among the tools not considered for implementing GCsnap2.0 is Numba, which is designed to improve the performance of computationally intensive parts through its JIT compiler. The only case where GCsnap would benefit is the many–against–many sequence search, where MMseqs already solves the computational requirements. Therefore, there is no need to consider Numba further. Parla is also discarded as a possible solution. The reason is that Parla is designed to manage heterogeneous resources on a single node. The nodes on which we want to execute gc2C are not of this type.

Writing scripts in programming languages other than Python to improve the performance of GCsnap was never part of this work, but using a tool based on it was considered. However, the problem with both PyCOMPS and Flux is that they require a running instance of the underlying infrastructure. For PyCOMPS, this means installing COMPS, including the Java development kit. Installation details are presented in [55]. Generally, users of HPC clusters do not have the means to install what they need. Therefore, deploying GCsnap on a cluster would be difficult and violate the portability requirement. Therefore, we will not consider PyCOMPS and Flux further. For the same reason, we discard ExaWorks. Since it is built on top of Flux, it requires the same infrastructure. This makes it difficult to deploy, which again violates the portability requirement.

The last tool shown in Figure 4.1 is Parsl. It is very similar to Dask in that it interacts with cluster management tools and uses concurrent.futures and mpi4py under the hood. Although it is suitable for this work, we did not consider it further. Parsl makes it convenient to build parallel workflows from existing Python code with its decorators. However, gc2D and gc2C are implemented from scratch, which mitigates this advantage. In addition, Dask offers additional features over Parsl, most notably support for distributed Pandas dataframes. All in all, this led to the decision to use Dask instead of Parsl. Experimenting with both tools was not considered an option to keep the amount of work manageable.

We now know which tools to consider for implementing gc2D and gc2C. However, we still need to decide which one to use. In order to base our decision on some empirical evidence, we have performed several experiments, which will be presented in the following section.

5.2 Parallel Tools for GCsnap2.0 Desktop

The first experiments were designed to gather information about the parallel tools under consideration for gc2D: Multiprocessing, concurrent.futures, and Dask's `Client`. We aimed to evaluate which works best for the first block of the GCsnap workflow (see Section 3.1). To achieve this, we designed two experiments:

1. Mapping between ID standards using the UniProt API. GCsnap1 requests information from the API separately for each target. For this experiment, we adapted the code to work with batches, allowing multiple arguments per request. Determining the optimal batch size was also part of the evaluation.

2. Find, download and extract assemblies. All the functionality of GCsnap1 to perform these steps is combined into a Python class, importing the parallel functionality from another module. However, the code itself is not yet optimized, for example by replacing loops with list comprehensions.

Both experiments were run with different parameters as presented in Table 5.2 and executed on sciCORE. The most significant improvement is that we implemented wrapper functions for each tool under evaluation. An example of the wrapper using `ProcessPoolExecutor` from concurrent.futures is shown in Code 5.1. All wrappers receive an iterable list of arguments, which is processed by the specified number of CPU cores, either as processes or threads.

Code 5.1: Example of the wrapper function using concurrent.futures' `ProcessPoolExecutor`.

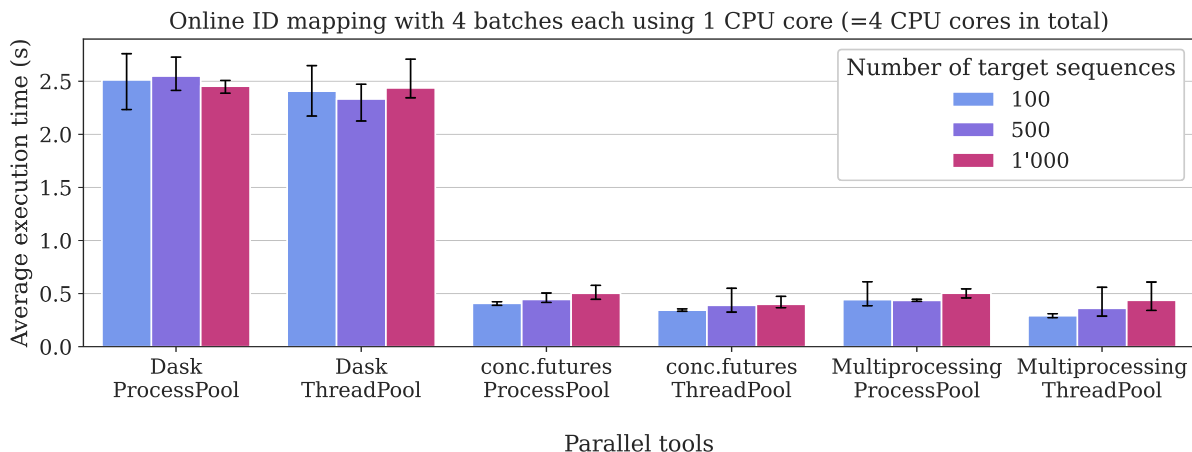
```python
from concurrent.futures import ProcessPoolExecutor
def futures_process_wrapper(n_processes: int, parallel_args: list[tuple], func: Callable) -> list:
    """
    Apply a function to a list of arguments using ProcessPoolExecutor. The arguments are passed as
    tuples and are unpacked within the function. As completed is used to get the results in the
    order they finish.

    Args:
        n_processes (int): The number of processes to use.
        parallel_args (list[tuple]): A list of tuples, where each tuple contains the arguments for
    the function.
        func (Callable): The function to apply to the arguments.

    Returns:
        list: A list of results from the function applied to the arguments in the order they
    finish.
    """
    with ProcessPoolExecutor(max_workers=n_processes) as executor:
        futures = [executor.submit(func, arg) for arg in parallel_args]
        result_list = [future.result() for future in as_completed(futures)]
    return result_list
```

Results of the online ID mapping experiment

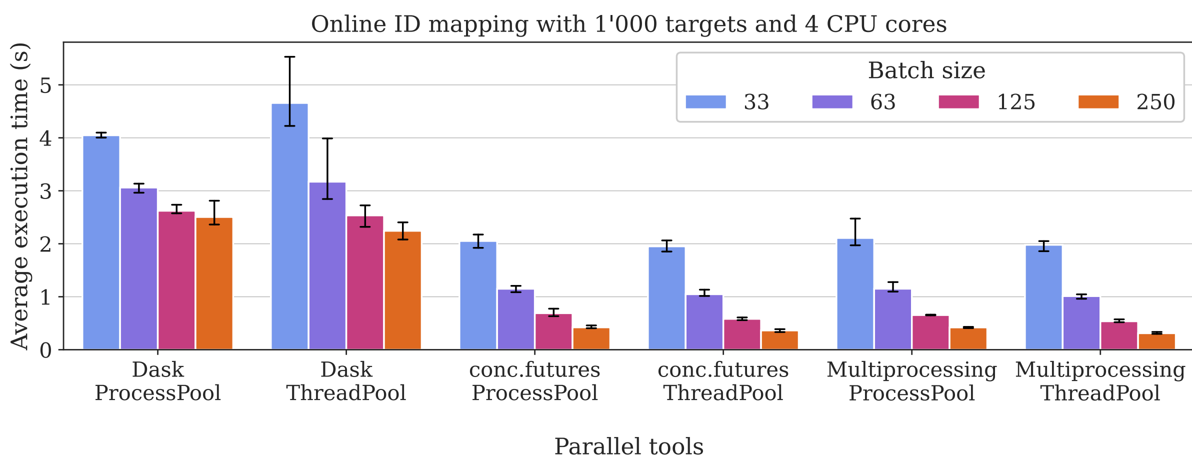
The results of this experiment are shown in Figure 5.1. Our goal is to determine which tool works best and what an optimal batch size would be. The first question is addressed in Figure 5.1a. Dask has the worst performance. This is due to the additional computation required to construct the task graph to be executed, which makes Dask less efficient compared to other tools. Moreover, threads appear to perform

Table 5.2: Design of factorial experiments to assess the different parallel tools considered for the implementation of GCsnap2.0 Desktop.

Factor	Value	Properties
Application	ID mapping online	Number of targets: 100, 500, 1'000 Number of CPU cores: 1, 2, 4, 8, 16 Parallel tools: Multiprocessing, concurrent.futures and Dask; each with threads and processes Batch size: 33, 66, 125, 250 Default: number of targets / cores
	Assembly handling online	Number of targets: 10, 20, 50 Number of CPU cores: 1, 2, 4, 8, 16 Parallel tools: Multiprocessing, concurrent.futures and Dask; each with threads and processes
Metrics	Program performance	Execution time (seconds)
Computing system	1 sciCORE node	2 Xeon E5-2630v4, 2.2 GHz; each with 10 cores and 25 MB L3 cache; 256 GB RAM; API-enabled connectivity
Validity	Repetitions	5



(a) The average execution time with various targets and 4 CPU cores, each running one batch. The batch size equals the number of targets divided by 4.



(b) The average execution time with different batch sizes and 1'000 input targets, executed on 4 CPU cores.

Figure 5.1: Average execution time of the online ID mapping experiment over 5 repetitions with different numbers of CPU cores, input targets, batch sizes, and parallel tools. The abbreviation conc.futures stands for the concurrent.futures module. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on 2 Xeon E5-2630v4 with a total of 20 CPU cores.

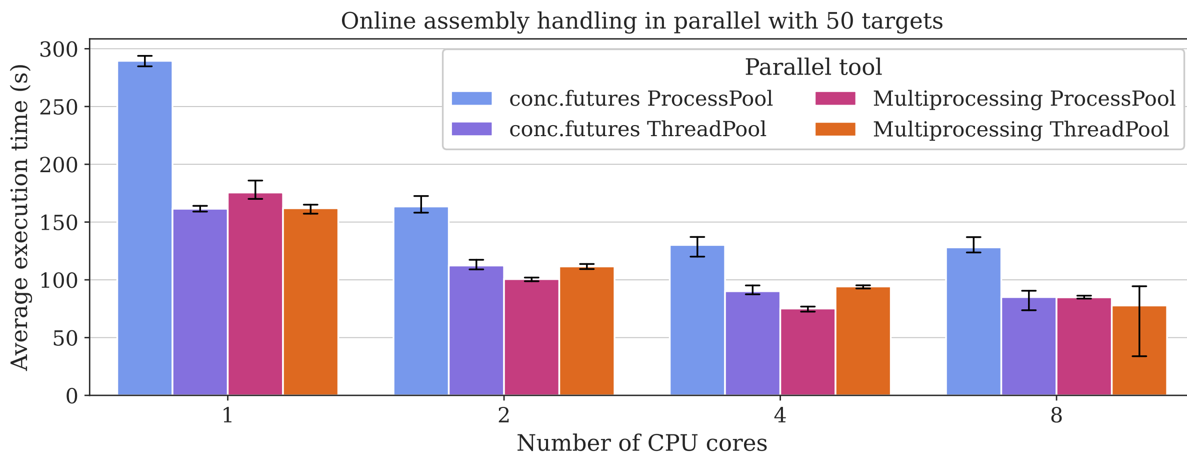
better relative to processes. The tools that work with threads are subject to the GIL, meaning they are not executed in parallel even when running on multiple CPU cores. However, efficient context switching allows overlapping of URL requests and leads to faster execution than with parallel processes.

Regarding batch size, Figure 5.1b shows the average execution time over 5 repetitions of each experiment. Two observations can be made. First, the previous finding that Dask is less efficient is confirmed, as Dask has a longer execution time than the other tools for each batch size. Second, larger batch sizes perform better. As the batch size increases for a given number of targets, fewer API requests are required. The results show that the UniProt API can handle larger requests efficiently, as doubling the batch size roughly halves the execution time. In fact, the UniProt API allows up to 100,000 target IDs to be submitted in a single request [56], therefore larger batch sizes are preferable.

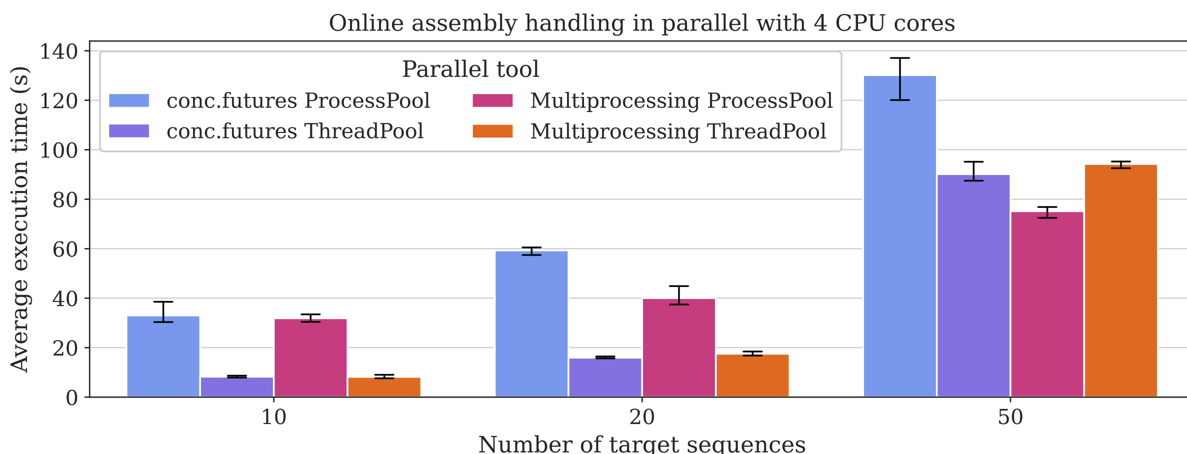
Results for the online assembly handling experiment

Unlike the previous experiment, this one not only requests data, but also parses the assembly files. Figure 5.2a shows the results. It is important to note that the actual height of the bars is not of interest, as the underlying code is still unoptimized. Nevertheless, The results indicate that the concurrent.futures module performs the worst. However, this finding is misleading.

The results are skewed by the way the work is sent to the processes. For concurrent.futures, our



(a) The average execution time for 50 targets.



(b) The average execution time when running on 4 CPU cores.

Figure 5.2: Average execution time of the online assembly handling experiment over 5 repetitions with different numbers of CPU cores, input targets, and parallel tools. The abbreviation conc.futures stands for the concurrent.futures module. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on 2 Xeon E5-2630v4 with a total of 20 CPU cores.

implementation submits one argument at a time, as can be seen in Code 5.1 line 15. This contrasts with Multiprocessing's `Pool`, which relies on the built-in `map` function to process the iterable in chunks. By default, the chunk size is not 1, but is calculated based on the length of the iterable and the number of CPU cores. The actual formula can be found in [57]. Using chunks is more efficient because it requires less communication with the processes in the pool. Since the chunk size is calculated using a heuristic, the programmer does not have to worry about workload allocation, which is convenient for an implementation of gc2D. Furthermore, Multiprocessing with processes outperforms Multiprocessing with threads for a larger number of targets, as shown in Figure 5.2b. This suggests that the advantage of overlapping URL requests with threads seen in the first experiment is lost when the computational part becomes dominant.

In conclusion, Multiprocessing's `ProcessPool` is best suited for gc2D, as the tasks involve collecting data through APIs and extracting relevant information. Furthermore, the automatic workload adjustment facilitates the implementation. With respect to APIs, larger batch sizes are preferred. Wrapping the invocation of the parallel tools in wrapper functions provides an easy and flexible way to call them whenever needed. This simplifies modularization.

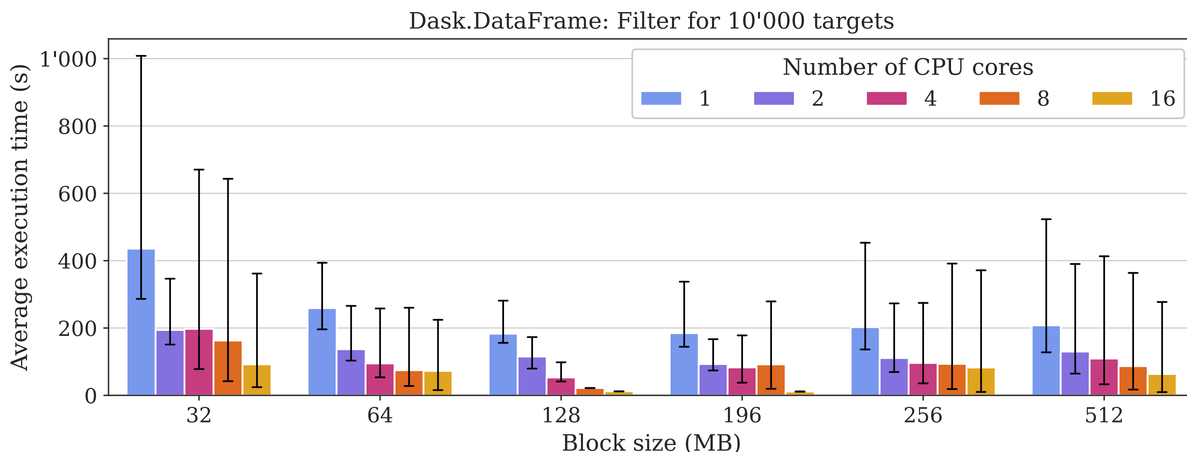


Figure 5.3: Average execution time of the ID mapping experiment with Dask.DataFrame over 5 repetitions with different numbers of CPU cores and block sizes with 10'000 targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on 2 Xeon E5-2640v4 with a total of 20 CPU cores.

5.3 ID Mapping with Dask.DataFrame

The goal is to evaluate the suitability of Dask.DataFrame for ID mapping in gc2C. The underlying data for the UniProt mapping API is available for download (see Section 6.4). Since the file is large and may not fit in memory, the idea is to use Dask.DataFrame to represent a Dask data frame as many partitions of Pandas data frames. In order to get the mapping between the ID standards of any target, the combined data frame is filtered using the appropriate Pandas syntax and Dask handles the filtering of each partition. Based on the examples in [58], we used a format called Parquet [59] to store our partitions. These are pre-created for fast access during computation. The most important parameter is the block size, measured in bytes, which determines the size of each partition. The smaller the block size, the more partitions are needed. Since the optimal block size is not known beforehand, we tested different values during the experiment. Table 5.3 presents the experimental design. The main conclusion is provided below, with additional plots in Appendix B.2.

The results of varying the block size and the number of CPU cores with 10'000 target sequences are shown in Figure 5.3. The fact that the mean is close to the bottom of the error bar representing the minimum execution time over the 5 repetitions indicates the presence of outliers. To identify the cause of this issue, we plotted the execution time for each repetition on 1 CPU core separately in Figure 5.4. Clearly, the first repetition is the outlier in all cases. The operating system keeps track of the partition locations on the network file system, meaning they are found faster when repeating the experiment. This

Table 5.3: Design of factorial experiments to assess the suitablity of *Dask.DataFrame* for an implementation of GCsnap2.0 Cluster.

Factor	Value	Properties
Application	ID mapping with Dask Dataframe	Number of targets: 10'000, 50'000, 100'000, 500'000 Number of CPU cores: 1, 2, 4, 8, 16 Block size (MB): 32, 64, 128, 196, 216, 512
Metrics	Program performance	Execution time (seconds) of parsing the input, read parquet files, and filter data
Computing system	1 miniHPC node	2 CPU Xeon E5-2640v4, 2.4GHz; each with 10 cores and 25 MB L3 cache; 64 GB RAM;
Validity	Repetitions	5

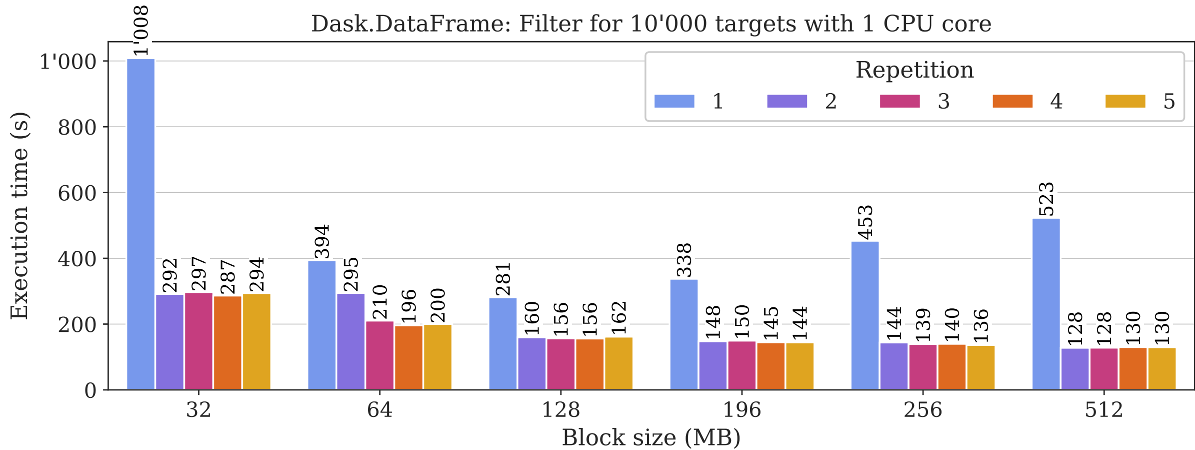


Figure 5.4: Execution time of each repetition of the ID mapping experiment with Dask.DataFrame with different block sizes, 10'000 targets and 1 CPU core. Experiments were conducted on 2 Xeon E5-2640v4 with a total of 20 CPU cores.

suggests that the first repetition is not the outlier, but rather the other 4 are atypically fast. Given the long execution time for filtering the ID mapping table with Dask.DataFrame on the first run, we do not consider this to be a suitable tool for implementing GCsnap2.0 Cluster.

5.4 Dask.distributed vs. MPI for Python

We then evaluate whether to use Dask.distributed or mpi4py for gc2C. As discussed in Section 4.2, using mpi4py involves controlling the master–worker relationship, which requires the programmer to include a control structure in the script to manage what is run on which MPI rank. This approach seems impractical for a large project like gc2C, which has many modules and dependencies between the modules. A solution is to use mpi4py.futures, which is designed to work with pools of processes, similar to concurrent.futures. Therefore, it can be bundled into a wrapper function. Since Dask and mpi4py.futures are equally easy to implement, the choice is based on performance.

Before discussing the experiments conducted, a few remarks regarding the deployment of Dask and mpi4py.future on an HPC cluster are necessary. Running gc2C in a distributed environment requires interaction with cluster management software. Both clusters we are working on, miniHPC and sciCORE, use SLURM [39]. Scripts are started by submitting a job script that requests cluster resources to run our application. One difference between the two is what resources are being requested. As mentioned in Section 4.2, Dask sends requests for additional worker nodes as needed from the Python script. So the main Python thread, which also manages the Dask scheduler, is started with a job script asking SLURM for a node, and Dask does the rest. This contrasts with mpi4py.futures, where all resources to be used are requested by the job script.

As a result, Dask needs an additional node to run the main thread and the scheduler. The situation when running on sciCORE is shown in Figure 5.5. This does not seem efficient, especially on busy HPC systems like the sciCORE cluster. A user must wait twice to get the resources, and while waiting for the workers, already blocks the resources of the node where the main thread is idle. A workaround to overcome this is mentioned in [60]. The main idea is to manually start the scheduler and the workers separately from the same job script. However, there is no way to use the node where the main thread is running as a worker [60].

The obvious approach to circumvent the problem was to run the Python script directly from the login node on the cluster, but this did not work. Dask workers could not connect to the Dask scheduler running on the login node. It is possible that the connection from the compute nodes to the login node in the cluster is blocked, as suggested in [60]. In any case, the solution to this problem is beyond our control and would require assistance from the cluster administrators.

Accepting the deployment issues as beyond our capabilities, we proceed to evaluate whether mpi4py.futures or Dask performs better. In [35], it was shown that mpi4py.futures outperforms Dask in most scenarios.

```
Tue Aug 13 16:50:55 2024
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2303424   scicore 1000_2_4 kruret00 PD       0:00      1 (Dependency)
           2303423   scicore 1000_2_2 kruret00  R       0:11      1 sca30

Tue Aug 13 16:51:00 2024
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2303701   scicore dask-wor kruret00 PD       0:00      1 (None)
           2303700   scicore dask-wor kruret00 PD       0:00      1 (None)
           2303424   scicore 1000_2_4 kruret00 PD       0:00      1 (Dependency)
           2303423   scicore 1000_2_2 kruret00  R       0:16      1 sca30

Tue Aug 13 16:51:05 2024
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2303701   scicore dask-wor kruret00 PD       0:00      1 (None)
           2303700   scicore dask-wor kruret00 PD       0:00      1 (None)
           2303424   scicore 1000_2_4 kruret00 PD       0:00      1 (Dependency)
           2303423   scicore 1000_2_2 kruret00  R       0:21      1 sca30

Tue Aug 13 16:51:10 2024
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2303701   scicore dask-wor kruret00 PD       0:00      1 (None)
           2303700   scicore dask-wor kruret00 PD       0:00      1 (None)
           2303424   scicore 1000_2_4 kruret00 PD       0:00      1 (Dependency)
           2303423   scicore 1000_2_2 kruret00  R       0:26      1 sca30

Tue Aug 13 16:51:15 2024
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2303701   scicore dask-wor kruret00 PD       0:00      1 (None)
           2303700   scicore dask-wor kruret00 PD       0:00      1 (None)
           2303424   scicore 1000_2_4 kruret00 PD       0:00      1 (Dependency)
           2303423   scicore 1000_2_2 kruret00  R       0:31      1 sca30

Tue Aug 13 16:51:20 2024
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2303424   scicore 1000_2_4 kruret00 PD       0:00      1 (Dependency)
           2303700   scicore dask-wor kruret00  R       0:01      1 sca36
           2303701   scicore dask-wor kruret00  R       0:01      1 sca37
           2303423   scicore 1000_2_2 kruret00  R       0:36      1 sca30
```

Figure 5.5: Screenshot of the terminal when executing Dask on sciCORE. The main Python scripts run on their own node, requesting additional worker nodes, which are assigned then.

To asses its suitability for our case, we conducted several experiments according to the design presented in Table 5.4. The task consists of opening and parsing assembly files. For convenience, the experiments were run on miniHPC, which is much less busy than sciCORE. The main results can be seen in Figure 5.6, with additional shown in Appendix B.3. Dask performs better in the single node experiment, while mpi4py.futures has a lower average execution time when using more than 2 nodes. There is no further

Table 5.4: Design of factorial experiments to compare the performance of Dask.distributed and mpi4py.futures.

Factor	Value	Properties
Application	Parsing assemblies with Dask.distributed	Number of targets: 1'000, 2'000, 5'000, 10'000 Number of compute nodes: 1, 2, 4, 8, 16 Number of CPU cores per node: 1, 2, 4, 8, 16
	Parsing assemblies with mpi4py.futures	Number of targets: 1'000, 2'000, 5'000, 10'000 Number of nodes: 1, 2, 4, 8, 16 Number of CPU cores node: 1, 2, 4, 8, 16
Metrics	Program performance	Execution time (seconds)
Computing system	miniHPC	Each node with 2 CPU Xeon E5-2640v4, 2.4GHz; each with 10 cores and 25 MB L3 cache; 64 GB RAM
Validity	Repetitions	5

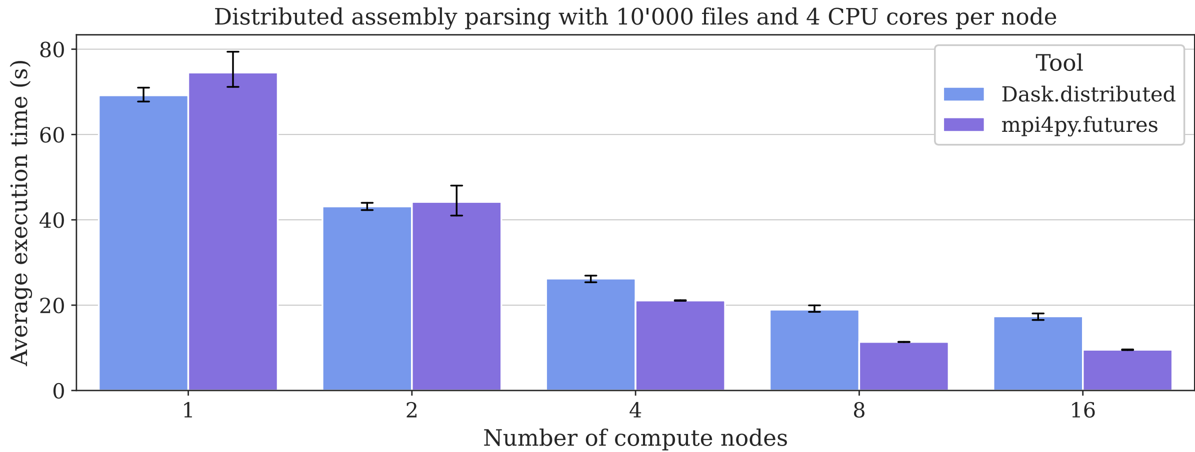


Figure 5.6: Average execution time of the assembly file parsing experiment with Dask.distributed and mpi4py.futures over 5 repetitions with different numbers of nodes, 4 CPU cores per node, and 10'000 files. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on miniHPC nodes, each with 2 Xeon E5-2640v4 and a total of 20 CPU cores.

improvement when using more than 8 nodes. However, this is due to the small size of the workload, which consists of 10,000 files to be parsed. Both tools have a coordination and communication overhead that dominates the computation of such a small workload. Since gc2C should perform well on many nodes, mpi4py seems to be the better choice. In summary, we will use mpi4py for our implementation of gc2C. It outperforms Dask when running on multiple nodes and does not have the problems when deployed on a SLURM managed cluster.

5.5 Summary of the Assessment

In this chapter, we discussed the advantages and disadvantages of the various tools presented and evaluated their suitability for implementing GCsnap2.0 Desktop and GCsnap2.0 Cluster with experiments. The most important aspects are:

- Parallelism in gc2D is based on `ProcessPool` from the Multiprocessing module. The experiments strongly suggest that it works best for tasks involving I/O operations and computations.
- Results of online experiments have shown that requesting information from APIs is most efficient when using large batch sizes.
- Writing wrapper functions that encapsulate parallel tools is feasible, simplifying implementation and modularization.
- Using the Dask.DataFrame module to work with partitioned data is not suitable. The first time an experiment is run on a node, it takes much longer than the subsequent runs. The execution time of the first run is considered to be too long and not applicable to real-world scenarios.
- We use mpi4py.futures to enable distributed computing for gc2C.

With these points in mind, the next chapter presents the new implementation of both applications.

Chapter 6

GCsnap2.0

This section covers the implementation of GCsnap2.0 and its two versions, GCsnap2.0 Desktop (gc2D) and GCsnap2.0 Cluster (gc2C). The first part presents the results from the user survey to gain a more profound understanding about the issues users faced with GCsnap1 and their suggestions for improvement. Thereafter, we present new features in GCsnap2.0 before detailing out the implementations of gc2D and gc2C, with a particular focus on the data that needed to be stored on the cluster.

6.1 Results of user survey

One of the goals of creating GCsnap2.0 was to increase the level of its user-friendliness. We conducted a user survey to get some insight into how GCsnap1 is being used, what could be improved, and what new features should be included,. The anonymous survey was open between April 16 and August 3, 2024 and consisted of 22 questions. The full questionnaire can be found in Figure A.1. Unfortunately, the number of responses was very low. Despite sending reminders and adding the link to the survey to the GCsnap1 repository on GitHub, only 2 people participated. This raises concern about the reliability and generalizability of the results. Nevertheless, some responses had a direct impact on our design decisions, while others highlighted problems with GCsnap1.

Questions related to GCSnap argument list
Surprisingly, not all respondents were aware that GCsnap1 supports 40 optional arguments. When asked about the most commonly used argument categories (see Figure A.1e), only 4 categories were mentioned. One reason may be the tedious nature of specifying so many arguments from a command line terminal. To simplify the process, GCsnap2.0 includes the ability to pass arguments via a configuration file, a feature welcomed by all respondents.

Parallel execution and workload
One of the objectives of this thesis is to ensure that the new implementation of GCsnap2.0 runs in parallel. However, it was unclear how many input sequences users typically use to run GCsnap1 and how much computing resources are needed for this. The question about the number of targets (see Figure A.1g) revealed a need to run GCsnap2.0 with up to 2'000 targets. As mentioned in Section 3.2, this takes a very long time and is not guaranteed to finish. Regarding computing resources, one user explicitly expressed the desire to run GCsnap2.0 on a cluster, while the other preferred the option to specify the number of CPU cores used.

Revealed issues
The most insights gained were from the open–ended questions, especially when users were asked about bugs and problems (see Figure A.1g). One issue mentioned is the reproducibility of GCsnap1. Occasionally it reports that an assembly cannot be found, but the same assembly is located when the application is rerun. Upon analyzing the code in detail, we discovered that online request errors are improperly handled. When an API limit is reached or a request times out, GCsnap1 simply excludes that target. Since such connection–related problems are sporadic, the results of two GCsnap1 runs can differ. Another aspect mentioned by respondents was missing output. The reason for this behavior was again due to a

```
--------------------------------------------
GCsnap
GCsnap is a python-based, local tool that generates interactive snapshots of conserved protein-coding genomic contexts.
Thanks for using it! 💙
--------------------------------------------
✅ Done configuration file config.yaml loaded
✅ Done parsing CLI arguments and config.yaml
✅ Done parsing targets
🔨 Working on Task target_sequences_100 with 100 targets
✅ Done mapping target sequences to UniProtKB-AC
✅ Done mapping target sequences to RefSeq
✅ Done mapping target sequences to EMBL-CDS
✅ Done all mapping done. Table saved to mapping.csv
      Assembly summary genbank is not older than 14 days, not downloading.
✅ Done parsing genbank assembly summary
      Assembly summary refseq is older than 14 days, downloading again.
      Downloading refseq assembl… ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
✅ Done downloading refseq assembly summary
✅ Done parsing refseq assembly summary
✅ Done find identical protein groups
✅ Done download assemblies and extract flanking genes
 ⚠ No flanking genes found for 13 target sequences. Check gcsnap_target_sequences_100.log
✅ Done download sequences
```

Figure 6.1: Screenshot of the terminal when running GCsnap2.0.

runtime error that was not handled carefully. When it occured, GCsnap simply stopped plotting without reporting the problem.

In addition to these runtime-related issues with GCsnap1, the survey responses also pointed to another problem. GCsnap1 is supposed to run on all major operating systems, including Windows [3, p. 6]. One respondent specifically mentioned using Windows Subsystem for Linux (WSL). Curious about why it was not executed directly on Windows, we attempted to run it ourselves and were unsuccessful. The main reason is the use of the Unix command *wget* to download files, which is not standard on Windows. Additionally, we could not find a working installation candidate of the protein family finding tool MMseqs2 for Windows [13]. Since portability is one of the goals of this work, finding solutions to these issues is essential when implementing GCsnap2.0.

6.2 New Features and Improvements

Informative terminal messages and logging

GCsnap1 had no real logging mechanism, relying instead on `print` statements, which flooded the terminal with messages. Logging was done by piping the terminal output to a file. One of the first new features was the separation of terminal output and logging. Since, user–friendliness is one of the goals of this work, we used the Rich module for Python [61] to have nicely formatted terminal messages. These messages inform the user about the state of execution and display warnings or errors. An example of this can be seen in Figure 6.1. The logging is based on Python's logging module [62], which is part of the standard Python library. During execution, warnings are written to a file. Most conveniently, a warning message including the target ID is logged if a target cannot be mapped or the corresponding assembly cannot be found.

Providing arguments through a configuration file

Entering up to 40 arguments when running GCsnap1 is tedious. We added the functionality to pass arguments through a configuration file named `config.yaml` to improve usability. This makes it easier to use GCsnap2.0, especially when many parameters differ from their default value. In this scenario, the user can modify them directly in the configuration file instead of passing them through the command line. Based on the survey responses, this feature is expected to be well–received by GCsnap users. During these changes, we have also improved the output of the `--help` from in the command window. The concise layout increases the information peovided, as shown in Figure 6.2. Our implementation uses the pyyaml module for Python [63].

```
(gcsnap) PS C:\MT\GCsnap> GCsnap --help
--------------------------------------------
GCsnap
GCsnap is a python-based, local tool that generates interactive snapshots of conserved protein-coding genomic contexts.
Thanks for using it! 💙
--------------------------------------------
✅ Done configuration file config.yaml loaded
✅ Done parsing CLI arguments and config.yaml

usage: GCsnap --targets <targets> [Optional arguments]

If CLI arguments not sepcified, default value from config.yaml
--overwrite-config to replace values with CLI Arguments
Or change them manually in config.yaml

-t, --targets
    List of input targets. Can be a list of fasta files,a list of text files encompassing a list of protein sequence identifiers,a list
    [config.yaml value: None]

Optional arguments:
--out-label
    Name of output directory. If default, name of the input file.
    [config.yaml value: default]
--tmp-mmseqs-folder
    The temporary folder to store mmseqs files. May be changed so that intermediary mmseqs files are saved somewhere else then the auto
    [config.yaml value: None]
--collect-only
    Boolean statement to make GCsnap collect genomic contexts only, without comparing them.
    [config.yaml value: False]
```

Figure 6.2: Screenshot of the terminal when using the `--help` command to get information about all arguments, allowed input, and defaults as defined in the configuration file.

Improved installation process and portability

Like GCsnap1, GCsnap2.0 relies on Conda environments [64] to manage software packages and Python libraries. Both the desktop and cluster applications of GCsnap2.0 are publicly available on GitHub [65, 66]. Installing GCsnap2.0 uses the `pyproject.toml` file which lists all dependencies. Unlike the original implementation, the new dependency specification includes not only the library names, but also the versions. This helps to avoid dependency issues and should simplify installation in the future. As one respondent noted, GCsnap1 did not run on Windows. In gc2D, files are downloaded using the Python libraries pypdl [67] and urllib.request [68] instead of Unix commands. The new implementation has been tested on Ubuntu, CentOS, MacOS and Windows to ensure portability. Problems have been reported when installing the necessary `gcc` library needed to build GCsnap2.0 on machines with ARM chips running MacOS, but those could be resolved manually and were not directly caused by GCsnap. The only inconvenience on Windows is that the user must specify the path to the downloaded static binary of MMseqs2 as an argument when running GCsnap2.0. Downloads are available in [13].

Advanced error handling

The error handling of GCsnap1 was suboptimal, as mentioned in Section 3.2 and Section 6.1. The new implementation improves error handling, particularly when working with online requests. When the request limit is reached, the server suggests the delay time. If no suggestion is provided, a random delay is used. The implementation is shown in Code 6.1.

Code 6.1: Example of improved error handling when retrieving data from NCBI Eutils including random back off when request limit is met and server does not send a delay suggestion.

```
initial_wait_time = 1  # Initial wait time in seconds
max_wait_time = 10  # Maximum wait time in seconds
while True:
    response = requests.get(url, timeout=timeout)
    if response.status_code == 200:
        xml_string = response.text
        return ET.fromstring(xml_string)
    elif response.status_code == 429: # limit hit
        # extract the time with wait suggestion from the response
        retry_after = response.headers.get('Retry-After')
        if retry_after:
            try:
                retry_after = int(retry_after)
            except ValueError:
                # If Retry-After is not a number, it might be a date
                retry_after_date = requests.utils.parse_date(retry_after)
                if retry_after_date:
                    wait_time = (retry_after_date - datetime.datetime.now()).total_seconds()
            wait_time = int(retry_after)
        else:
            # Random
            wait_time = random.uniform(initial_wait_time, max_wait_time)
        time.sleep(wait_time)
    else:
        raise WarningToLog('Eutils request failed with status code {}'.format(response.status_code
    ))
```

6.3 GCsnap2.0 Desktop

The application combines 26 different scripts and a total of 30 Python classes. The schematic view of all modules, dependencies and the main workflow is shown in Figure 6.3. The colors of the modules refer to the type of underlying functionality:

- Gray: Input
- Blue: Modules that are executed sequentially, without taking advantage of any parallelism.
- Yellow: Modules that use parallelism. As noted in Section 5.5, this is enabled by `ProcessPool` from Multiprocessing, encapsulated inside a function. See below for an explanation.
- Red: Modules that interact with online APIs to retrieve data, using batches of IDs for each request.
- Purple: Modules that provide general functionality. Some of them are direct dependencies of others, indicated by an arrow. Most other modules import the 3 modules on the top right. For simplicity, dependencies are not shown.

Modularization has two advantages. First and foremost, it increases maintainability and extensibility, meaning that new functionality within classes or new modules can be added with little effort. Second, the modules can be used stand alone in other projects.

The most important feature of gc2D is process parallelism. We use the wrapper function shown in Code 6.2. The wrapper takes a list of elements and a function as input. The `map_async` command applies the callable to each item in the list in chunks. If desired, the number of chunks can be controlled by passing a list of lists to the wrapper. If the iterable has as many elements as there are processes in the pool, it corresponds to chunk size of 1. But as stated in Section 5.2, this is not necessary given the chunk size computation heuristic of Multiprocessing. Most functions in gc2C take multiple arguments instead of just one. We can use the same wrapper, but combine all the arguments into a tuple, passing a list of tuples as input to the wrapper. To assure that all processes have finished, we use `pool.join` to synchronize before collecting the results. The wrapper returns a list of lists, where each sublist is the output of one process.

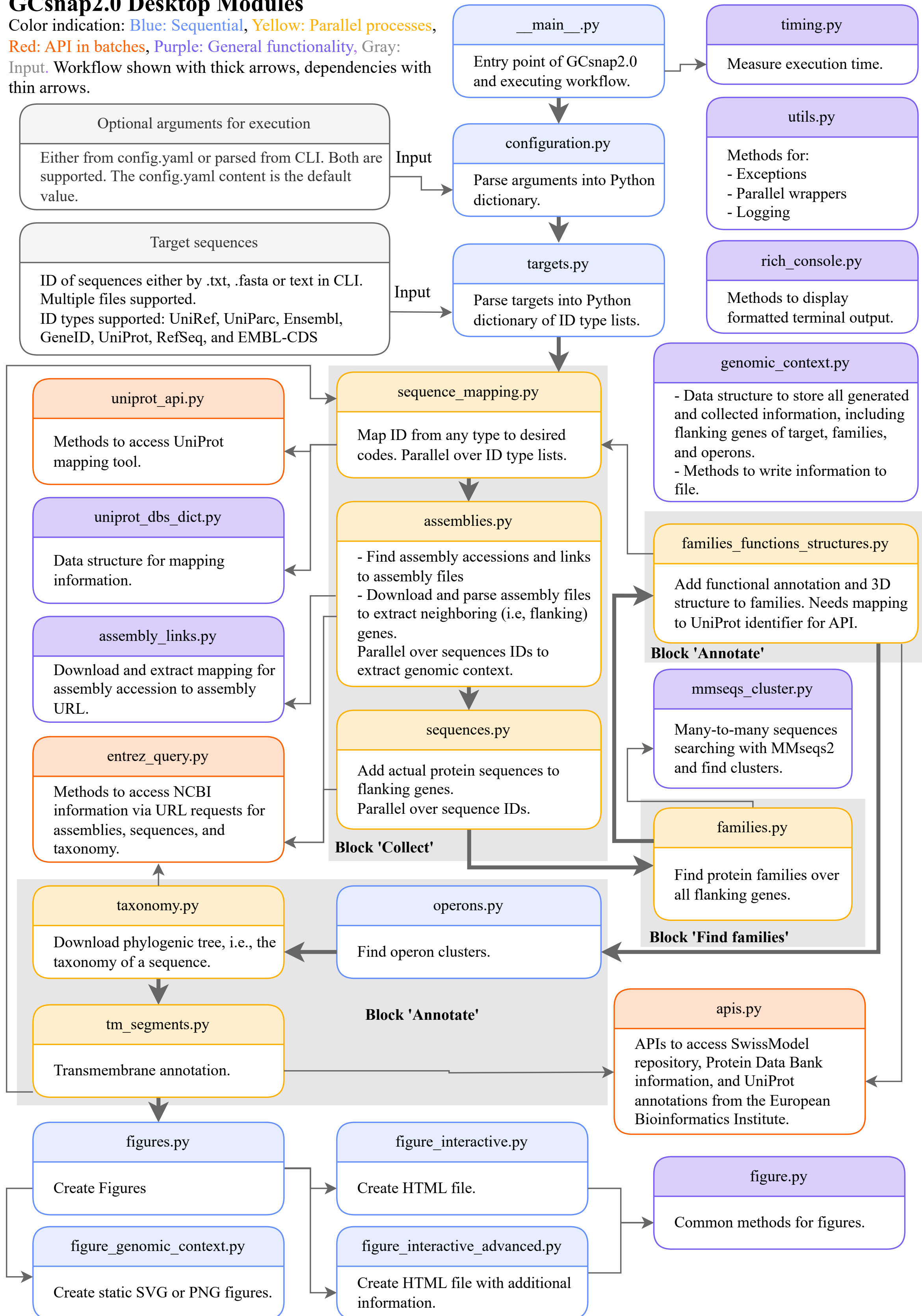


Figure 6.3: Modules and dependencies of GCsnap2.0 Desktop. Color indication: Blue are sequential modules, yellow are modules using parallel processes, red represents API request with batches, purple are modules of general functionality, and gray is the input. The main workflow is indicated by thick arrows, while dependencies are indicated by thin arrows.

Code 6.2: Example of the wrapper function using Multiprocessing's `ProcessPool`.

```
1  from multiprocessing import Pool as ProcessPool
2  def processpool_wrapper(n_processes: int, parallel_args: list[tuple], func: Callable) -> list:
3      """
4      Args:
5          n_processes (int): The number of processes to use.
6          parallel_args (list): A list arguments for the function.
7          func (Callable): The function to apply to the arguments.
8
9      Returns:
10         list: A list of results from the function applied to the arguments in the order they
       finish.
11     """
12     # create the pool
13     pool = ProcessPool(processes = n_processes)
14     # start execution
15         # map: Takes only one argument, hence upacking within the callable
16         # async: Results returned in order they finish
17     results = pool.map_async(func, parallel_args)
18     # Close pool to receive further work
19     pool.close()
20     # Synchronization: Wait until all have finished
21     pool.join()
22     # Get results
23     result_list = results.get()
24     return result_list
```

6.4 GCsnap2.0 Cluster

All data must be available on the cluster where gc2C is executed. We distinguish between two types. First, raw files that are parsed to extract information. For example, the assembly files from which the flanking genes are extracted. Second, databases to mimic the APIs used in gc2D. While UniProt provides a single file containing all the mappings between ID standards, NCBI does not supply such a combined summary. For example, the mapping from GenBank or RefSeq ID to assembly accession is not publicly available. Therefore, we extracted this information from the raw files and stored it in a database for quick retrieval during execution. Similarly, coding sequences were parsed from the available protein sequence files. As a database tool, we use SQLite with its Python implementation because it is a lightweight solution that does not require running a separate server [69]. Below we provide some details about the data, its source, the method to download it, and the means of storage on sciCORE. The number of files and the reported file size were measured using the `ls` command.

1. UniProt mapping between ID standards:
   - Source: https://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/idmapping/idmapping_selected.tab.gz
   - Number of files: 1
   - Size: 44 GB decompressed
   - Retrieved how: Manually download on Jul. 24, 2024
   - Kept as: SQLite DB mappings.db (65 GB) created with script available in [66]
2. NCBI summary tables for GenBank assembly files:
   - Source: https://ftp.ncbi.nlm.nih.gov/genomes/genbank
   - Number of files: 1
   - Size: 991 MB
   - Retrieved how: Manually download on Mar. 18, 2024
   - Kept as: TXT file
3. NCBI summary tables for RefSeq assembly files:

  - Source: https://ftp.ncbi.nlm.nih.gov/genomes/refseq
  - Number of files: 1
  - Size: 160 MB
  - Retrieved how: Manually download on Mar. 23, 2024
  - Kept as: TXT file

4. GenBank assembly files (_genomic.gff.gz):
   - Source: https://ftp.ncbi.nlm.nih.gov/genomes/all/GCA
   - Number of files: 1'678'176
   - Size: In total 461 GB compressed; File range: <1 KB, 274 KB, 188 MB (min, average, max)
   - Retrieved how: Synchronized with `rsync` in Mar. 2024
   - Kept as: GFF.gz

5. RefSeq assembly files (_genomic.gff.gz):
   - Source: https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF
   - Number of files: 362'140
   - Size: In total 138 GB compressed; File range: <1 KB, 380 KB, 79 MB (min, average, max)
   - Retrieved how: Synchronized with `rsync` in Mar. 2024
   - Kept as: GFF.gz

6. Mapping between GenBank and RefSeq IDs, assembly accessions, assembly files, and taxonomy ID:
   - Source: 2., 3., 4., and 5.
   - Number of files: 1
   - Size: 397 GB
   - Kept as: SQLite DB assemblies.db created with script [66]

7. GenBank protein sequence files (_protein.faa.gz):
   - Source: https://ftp.ncbi.nlm.nih.gov/genomes/all/GCA
   - Number of files: 1'676'631
   - Size: In total 1.29 TB compressed; File range: <1 KB, 770 KB, 304 MB (min, average, max)
   - Retrieved how: Synchronized with `rsync` in Mar. 2024
   - Kept as: SQLite DB sequences.db created with script [66]

8. RefSeq protein sequence files (_protein.faa.gz):
   - Source: https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF
   - Number of files: 362'090
   - Size: In total 289 GB compressed; File range: <1 KB, 780 KB, 29 MB (min, average, max)
   - Retrieved how: Synchronized with `rsync` in Mar. 2024
   - Kept as: SQLite DB sequences.db created with script available in [66]

9. NCBI taxonomy (rankedlineage.dmp in new_taxdump.tar.gz):
   - Source: https://ftp.ncbi.nih.gov/pub/taxonomy/new_taxdump
   - Number of files: 1
   - Size: 324 MB decompressed
   - Retrieved how: Manually download and extracted on Aug. 20, 2024
   - Kept as: DMP

The sequences.db is 366 GB in size, but only 4 percent of the records actually have values for the actual coding sequence. This corresponds to 250 million records. The rest have an empty string in the sequence field. The entire database filled with all coding sequences has an estimated size of 1.9 TB. 303 GB for the database without filled coding sequence fields, but including the index for quick access, and 1.6 TB for all data. To save space, we decided to limit the content to a subset of the sequences needed to evaluate gc2C.

This decision also saves time. The entire process of retrieving all the information and creating the restricted databases takes 17 days. 2 days of which are spent on the insertion of the 4 percent of the coding sequences. In a production environment where data should be updated regularly, the current data workflow is not efficient, although the frequency of such updates is debatable and depends on updates to the underlying data. While new RefSeq information becomes available daily [70], UniProt releases new versions every 8 weeks [71].

The amount of space and time required suggests that the current approach using SQLite databases is inappropriate for using GCsnap2.0 Cluster in a productive environment. Another limitation is that the data needed for functional annotation is not available for download. The solution is for the user to pass a JSON file containing the desired information as an argument to gc2C, which will be used to annotate functions and identify TM segments.

The modules and workflow of gc2C are shown in Figure 6.4. Although the structure is similar to the modularization of gc2D in Figure 6.3, the implementation is different, especially for the collect block. These modules rely on database handlers, shown in pink, to select information from the databases. In addition, the parallel functionality is provided by the class shown in red. Like gc2D, the parallelization is provided through a callable wrapper function that uses mpi4py.futures.

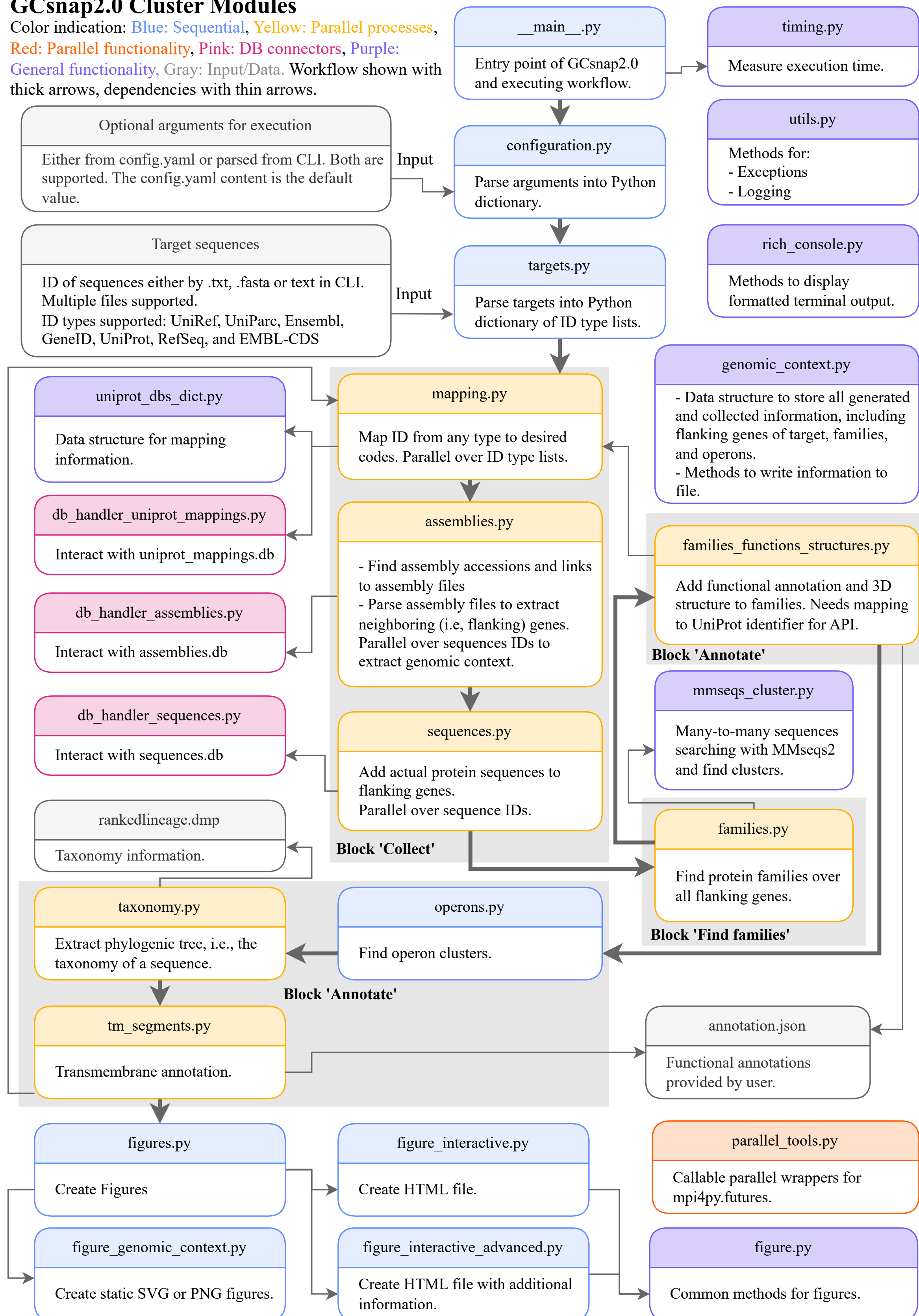


Figure 6.4: Modules and dependencies of GCsnap2.0 Cluster. Color indication: Blue are sequential modules, yellow are modules that use parallel processes, and red represents the module that contains the functionality to enable parallelism. Pink indicates the modules that interact with the databases, purple are modules of general functionality, and gray marks the input and necessary data. The main workflow is indicated by thick arrows, while dependencies are indicated by thin arrows.

Chapter 7

Evaluation

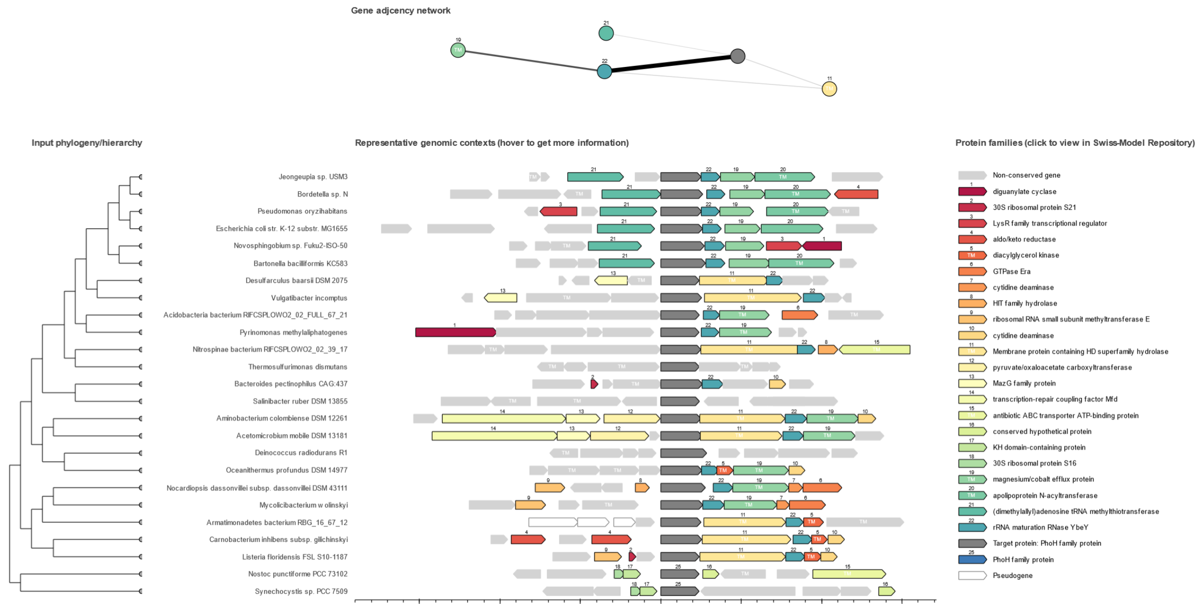
This chapter evaluates our implementation of GCsnap2.0 Desktop (gc2D) and GCsnap2.0 Cluster (gc2C). We begin by comparing the results of GCsnap1 with the output of GCsnap2.0. Next, we analyze the performance of gc2D to determin if the objective to improve the performance has been met. Finally, we present the results of running gc2C on sciCORE.

7.1 Consistency of GCsnap2.0

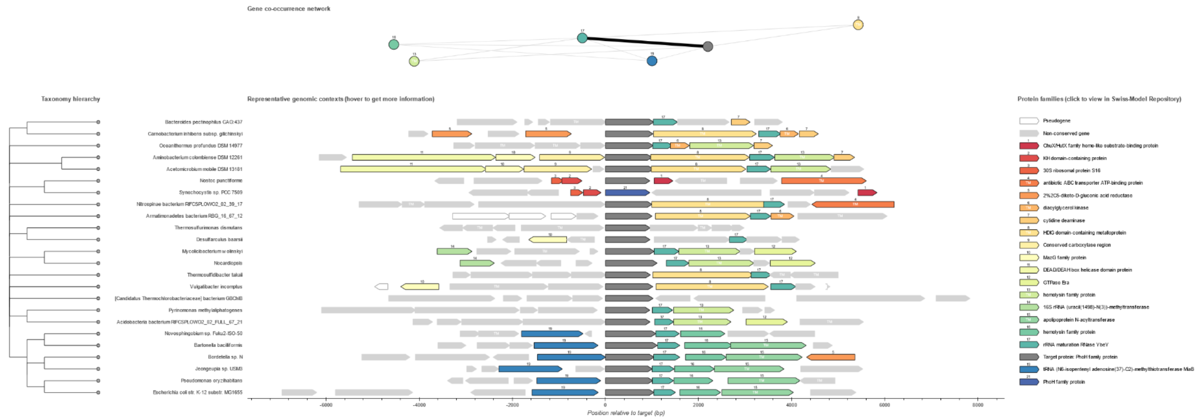
Ensuring that GCsnap2.0 replicates the functionality of GCsnap1 was an iterative process. It involved deploying the new implementation to selected day-to-day users, analyzing their feedback, and addressing any issues discovered. The consistency assessment was based on two pillars: (i) systematic comparison of the generated TXT and JSON files, and (ii) review of the graphical output. This process will continue beyond the submission of this thesis. Nevertheless, the current state is not only satisfactory, but

Table 7.1: Design of factorial experiments to evaluate the performance of GCsnap2.0 Desktop (gc2D) and the scalability of GCsnap2.0 Cluster (gc2C). Color indicates which experiment the information refers to.

Factor	Value	Properties
Application	GCsnap2.0 Desktop	Number of targets: 10, 20, 50, 100, 200, 500, 1'000, 2'000 Number of CPU cores: 1, 2, 4, 8, 16, 32, 64 Runtime arguments for gc2D: --annotate-TM True
	GCsnap2.0 Cluster	Number of targets: 10'000 Number of nodes: 2, 4, 8 Number of MPI ranks per node: 8 Runtime arguments for gc2C: --annotate-TM False --functional-annotation-files-path None
Metrics	Program performance	Execution time (seconds) end-to-end, of the three GCsnap task blocks, and individual steps
Computing system	1 miniHPC node	2 AMD EPYC 7742, 2.25 GHz; each with 64 cores and 256 MB L3 cache; 1'500 GB RAM; API-enabled connectivity
	sciCORE	Each node with 2 AMD EPYC 7742, 2.25 GHz; each with 64 cores and 256 MB L3 cache; 512 GB RAM
Validity	Repetitions	5, 1



(a) Screenshot of the interactive HTML output of GCsnap1, taken from the example *ybez_KHI* available in [2].



(b) Screenshot of the interactive HTML output of gc2D , created by reproducing the example *ybez_KHI* available in [2].

Figure 7.1: Comparison of the interactive HTML output from (a) GCsnap1 and (b) GCsnap2.0 Desktop (gc2D).

demonstrates exemplary consistency.

We replicated the example *ybez_KHI* available in [2] using gc2D, to compare the two applications. The interactive HTML output of GCsnap1 and GCsnap2.0 is shown in Figure 7.1. At first glance, they appear different, and they are, but for explainable reasons. The most important difference is the use of MMseqs instead of BLAST. As a result, different families were identified and the family network changed. The underlying data was also updated, especially for the taxonomy, resulting in a revised phylogeny. Despite these differences, the test users approved the results based on their expertise, demonstrating success in achieving consistency.

7.2 Performance of GCsnap2.0 Desktop

To evaluate the performance of gc2D and to compare it with GCsnap1, the experiments presented in Section 3.2 were repeated with the new tool. The design of the factorial experiment is shown in Table 7.1. As with GCsnap1, not all runs of gc2D were completed. The list of all 18 failed experiments can be found in Appendix B.4, along with additional plots. Unlike before, we now know the errors thanks to the updated error handling mechanism implemented. All failures occurred when using the UniProt API to map sequences or retrieving NCBI data, so we assume that temporary network interruptions are the

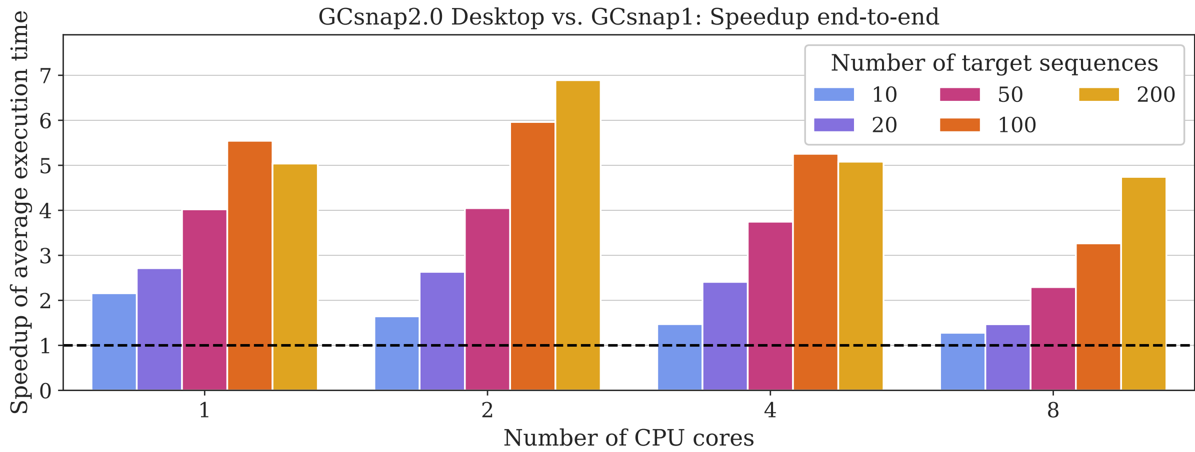


Figure 7.2: Speedup of the average end-to-end execution time over 5 repetitions of gc2D compared to GCsnap1 with different numbers of CPU cores and input targets. A value above 1 (black dashed line) means that gc2D had a lower average execution time. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished.

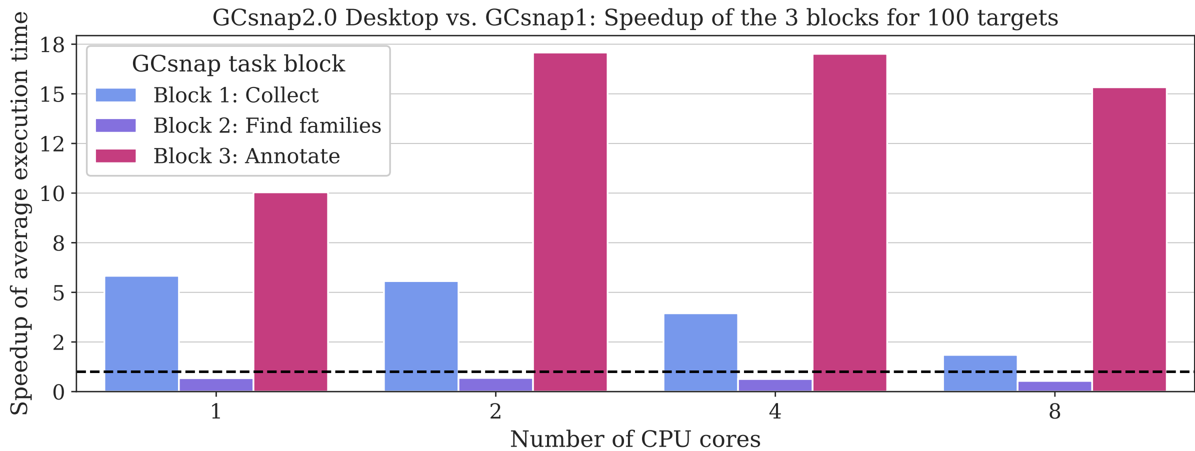


Figure 7.3: Speedup of the average end-to-end execution time over 5 repetitions of gc2D compared to GCsnap1 with different numbers of CPU cores and 100 targets for the 3 blocks of. A value above 1 (black dashed line) means that gc2D had a lower average execution time. Experiments were performed on a 2 AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished.

cause of these failures.

Figure 7.2 depicts the speedup calculated as the average execution time of GCsnap1 divided by the average execution time of gc2D. The numerator and denominator are the averages over the 5 repetitions of each experimental factor combination. A value greater than 1 means that gc2D was faster. The plot shows that GCsnap2.0 Desktop outperforms GCsnap1 in all cases. Additionally, the speedup is increasing as the number of input targets increases.

To analyze which block of GCsnap experiences the largest performance gain, we plotted the speedup of the 3 blocks of GCsnap in Figure 7.3. The new implementation of finding families with MMseqs is slower than in GCsnap1. Since we did not change anything in the actual execution of MMseqs, we suspect that the modularization is the cause, as importing the modules takes time and initializing the classes requires additional memory. However, this is not a concern as this step accounts for a small fraction of the end-to-end execution time (see Section 3.2). The other two blocks experience notable performance gain. The speedup of the annotation part is above $10x$ for different number of CPU cores.

The speedup for the steps of the annotate block is presented in Figure 7.4. We see that annotating functions, the taxonomy, and especially transmembrane annotation perform better. This is a direct result of requesting data from APIs in batches. The poor performance of the operons may again be

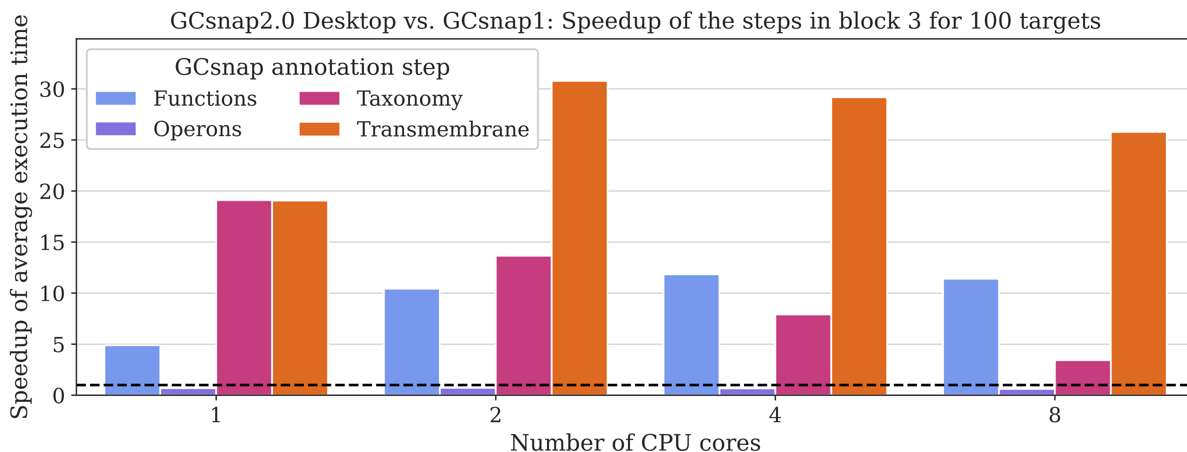


Figure 7.4: Speedup of the average end-to-end execution time over 5 repetitions of gc2D compared to GCsnap1 with different numbers of CPU cores and 100 targets for the annotation steps of task block 3. A value above 1 (black dashed line) means that gc2D had a lower average execution time. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished.

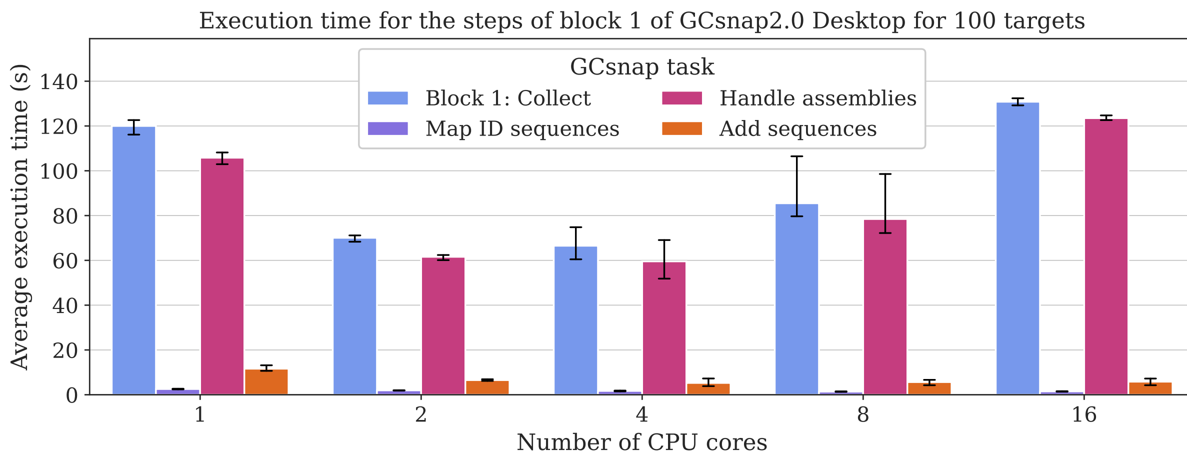


Figure 7.5: Average execution time of the first block and its steps of GCsnap2.0 Desktop over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores.

due to modularization, as this part is executed sequentially and the logic has not been changed. This underlies the success of gc2D as performance has been improved despite the modularization overhead.

To our surprise, the performance gain of collecting data in Figure 7.3 was not pronounced, especially, since the focus was on improving this part of GCsnap, given that collecting all necessary information accounted for the largest share of the end-to-end execution time (see Section 3.2). In order to analyze the steps of block 1 in detail, the average execution time over 5 repetitions of each experiment with gc2D is shown in Figure 7.5. Plotted is the average execution time of the block collect and of its three steps ID mapping, assembly handling, and adding the coding sequences. It is clear that handling the assemblies remains inefficient. Despite being executed in parallel, downloading the assembly files does not improve. On the contrary, it worsens with more computing resources. The reason is the limited bandwidth of the network. The more processes share the bandwidth, the slower each process becomes. This is less problematic for APIs as the amount of data transferred is small, but, as seen in Section 6.4, assembly files can be large. In conclusion, this part of GCsnap is only as fast as the download speed the connection allows and cannot be optimized further on a single node.

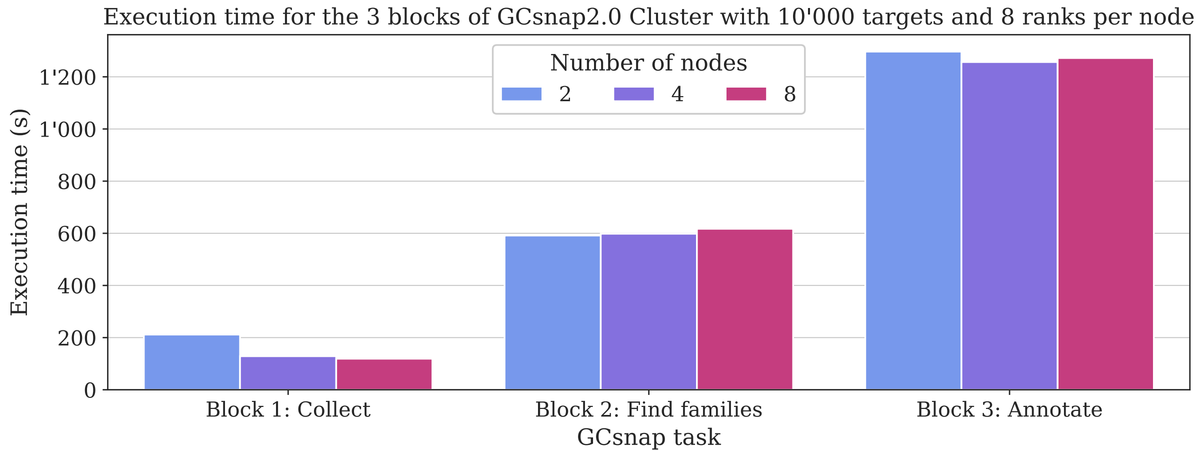


Figure 7.6: Execution time of the 3 blocks of GCsnap2.0 Cluster with different numbers of nodes, 10'000 input targets and 8 MPI ranks per node. Experiments were conducted on sciCORE nodes, each with 2 AMD EPYC 7742 with 128 CPU cores.

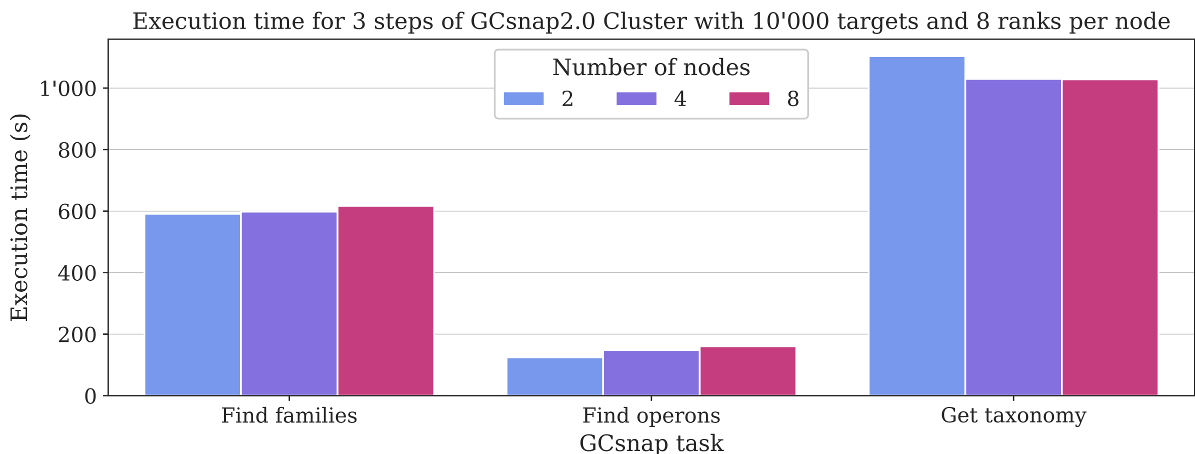


Figure 7.7: Execution time of the 3 steps of GCsnap2.0 Cluster that have the longest execution time with different numbers of nodes, 10'000 input targets and 8 MPI ranks per node. Experiments were conducted on sciCORE nodes, each with 2 AMD EPYC 7742 with 128 CPU cores.

7.3 Cluster scalability

To evaluate the performance of GCsnap2.0 Cluster, we use a real–world example from the Protein Universe Atlas mentioned in Section 1.1. The experimental design is presented in Table 7.1. Since we do not have a prepared functional annotation file, the corresponding runtime argument is set to `None`. We also disable TM annotations because they also rely on additional information passed to gc2C. These experiments were only repeated once, which raises concerns about the reliability of the results. However, they provide enough insight to evaluate our implementation.

Figure 7.6 shows the execution time of the 3 tasks block of GCsnap with different number of nodes and 8 MPI ranks per node. It is clear that our implementation has its limits. While block 1 does scale, the other 2 blocks do not. This is surprising since they all rely on the same parallelization functionality. We suspect that this is due to inefficient workload distribution. Unlike gc2D, where the Multiprocessing module has a heuristic to adjust the workload, mpi4py.futures does not and sets the chunk size to 1 by default. We approach this by passing a list of lists to the parallel wrapper function to ensure that each rank has a similar workload. However, the load balancing aspect reveals another limitation of our work. We did not use a profiling or tracing tool such as Score-P [72] to analyze our implementations in terms of memory efficiency or CPU usage.

We plotted the 3 steps that have the longest execution time in Figure 7.7 to determine which module

is responsible for the bad performance. The tree steps Find Families, Identify Operons, and Collect Taxonomy account for more than 80% of the total execution time of gc2C. Interestingly, all three were never considered a bottleneck, on the contrary, finding families had the best performance in GCsnap1. One possible explanation is that MMseqs does not run in parallel because it does not have access to the computational resources specified in the SLURM job script. But there is another reason. The tree steps mentioned were never considered a performance issue, and thus were not fully optimized by replacing all the original code.

On the positive side, GCsnap2.0 Cluster is able to process 10'000 targets in less than 40 minutes. This would never have been possible with GCsnap1. In conclusion, the identified bottlenecks of GCsnap1 have been resolved, revealing previously unknown limitations that can be addressed in future work.

Chapter 8

Conclusion

In this thesis, we propose a solution to reduce the execution time of GCsnap1, a genomic context analysis tool written in Python. The need to run on both local desktops and computer clusters required the writing of two versions: GCsnap2.0 Desktop (gc2D) and GCsnap2.0 Cluster (gc2C). While the former requests the necessary data from online sources via APIs, the latter uses previously stored information. We identified the data collection as the clear bottleneck of GCsnap1 and the cause of its poor performance. The large amount of data required for gc2C raises the question of how to manage the underlying data efficiently.

Both implemented versions are fully modularized, making it possible to use GCsnap functionality in other projects. In addition, we have improved several aspects of GCsnap. Clear and concise warning messages, improved logging, and the ability to pass arguments to GCsnap2.0 through a configuration file enhance usability. A precise definition of the building dependencies together with ensuring that GCsnap2.0 runs on all major operating systems ensures portability. However, the main focus was to increase performance by exploiting parallelism.

The limiting factor when running Python in parallel is the Global Interpreter Lock (GIL), a safety mechanism that prevents more than 1 Python thread from running at the same time. There are a variety of modules and libraries to overcome the Global Interpreter Lock (GIL), ranging from basic modules that work with processes from the standard Python library to powerful third–party tools designed for use on High Performance Computing (HPC) clusters. In order to assess which existing solutions are best suited for our work, we conducted a series of experiments to collect empirical data as a basis for decision making.

A comparison of the modules from the standard Python library showed that the `ProcessPool` functionality of Multiprocessing is best suited for gc2D. It is able to handle online queries and computations as well as working with processes. In addition, the automatic heuristic to determine the size of the workload sent to the participating process is a convenient feature because the programmer does not need to worry about workload allocation. The results of the primary analysis also showed that API requests are most efficient when done in batches.

The results of the experiment comparing Dask and mpi4py.futures showed that the latter is more suitable for gc2C, as the performance was better when running on more than 2 nodes. Furthermore, the way Dask interacts with SLURM when deployed on a cluster does not seem efficient, as a user has to wait twice for the requested resources. Once for the main thread to start and then for SLURM to allocate the additional requested worker nodes.

The evaluation of the new application shows that gc2D outperforms GCsnap1 in all experiments, but also reveals that the network connection is the determining factor for further performance gains. Requesting information from APIs and downloading many files can only be as fast as the download speed of the network connection. Regarding gc2C, the few experiments revealed that our implementation has some problems. On the one hand, it is possible to analyze thousands of sequences in a reasonable time. On the other hand, our implementation of certain modules is not efficient. In particular, those steps that were not a bottleneck in GCsnap1 leave room for additional improvement.

In summary, this work provides a user-friendly solution for fast genomic context analysis on local machines with GCsnap2.0 Desktop, and a tool for performing studies of entire protein families on SLURM–managed HPC clusters with GCsnap2.0 Cluster.

8.1 Future Work

Since GCsnap is freely available and actively used in daily operations, its development is continuous. This section outlines future work and possible adaptations, divided into short– and long–term goals. The aim is to ensure that the tool remains effective and adapts to evolving dependencies and future requirements.

Short-term adaptions
With the release of the newer, faster version of GCsnap, we anticipate a broader user base and, consequently, more but reports. Regularly addressing the issues to improve the quality will be a significant part of the future work, but not the only focus.

- As noted in Section 3.1, the currently supported version of TMHMM 2.0 is obsolete [20]. We need to add support for the new version DeepTMHMM [21]. Since this is a dependency rather than part of the implementation, updating the installation instructions should be sufficient. However, before proceeding, some analysis of the new tool and testing of interoperability with GCsnap2.0 is necessary.
- All GCsnap2.0 code is available on GitHub. Currently, gc2D and gc2C reside in our forked repository on two different development branches. They need to be merged into the original GCsnap repository. Obviously, both versions cannot coexist in the main branch together. The question of how and where gc2D and gc2C will reside needs to be addressed in the near future.
- The performance of gc2C is limited by three specific modules. Solving the unknown problems requires a thorough analysis to identify the cause. Solving these problems would increase the overall performance of gc2C, allowing the study of tens of thousands of protein sequences.

Long-term improvements
In the long term, there are two ultimate goals that would further enhance the usability of GCsnap2.0:

- Currently, it is possible to use the modules of GCsnap2.0 Desktop in a standalone manner. However, users still need to build GCsnap in a Conda environment, as there is no installation candidate. Making gc2D a Python package installable via the Python Package Index (PyPI) would allow users to include it in their projects and use individual modules easily. The challenge is whether and how to include the GCsnap workflow in the package. One approach could be to create a package that does not include the main module, and then create a new script that imports the GCsnap package and combines the classes to run the workflow.
- The summary of the data needed to run GCsnap2.0 Cluster in Section 6.4 highlighted that a large amount of data must be downloaded in advance, and databases need to be created. Regular updates are necessary to ensure the information remains up–to–date. Implementing a fully automated data pipeline to ensure the data remains up–to–date is the solution. Parts of the scripts executed to handle the data in this thesis can be adapted to implement such a data workflow, but significant improvements are needed to make the data pipeline efficient. Finally, the underlying database solution needs to be reviewed. While the lightweight SQLite was appropriate for this work, there are better options for a data pipeline that could reduce update time and generally increase the performance of gc2C.

Bibliography

[1] K. Mavromatis, K. Chu, N. Ivanova, S. D. Hooper, V. M. Markowitz, and N. C. Kyrpides, "Gene Context Analysis in the Integrated Microbial Genomes (IMG) Data Management System," *PLoS ONE*, vol. 4, Nov. 2009. doi: 10.1371/journal.pone.0007979.

[2] J. Pereira, "GCsnap GitHub Repository." [Online]. Available: https://github.com/JoanaMPereira/GCsnap. Accessed 21 Aug. 2024.

[3] J. Pereira, "GCsnap: Interactive Snapshots for the Comparison of Protein-Coding Genomic Contexts," *Journal of Molecular Biology*, vol. 433, no. 11, art. no. 166943, 2021. doi: 10.1016/j.jmb.2021.166943.

[4] SIB Swiss Institute of Bioinformatics, "Protein Universe Atlas." [Online]. Available: https://www.expasy.org/resources/protein-universe-atlas. Accessed 20 Aug. 2024.

[5] D. L. Nelson and M. M. Cox, *Lehninger Principles of Biochemistry: 6th International Edition.* Macmillan, 2012.

[6] Q. Xu and R. L. Dunbrack, "Principles and characteristics of biological assemblies in experimentally determined protein structures," *Current Opinion in Structural Biology*, vol. 55, pp. 34–49, Apr. 2019. doi: 10.1016/j.sbi.2019.03.006.

[7] A. E. Osbourn and B. Field, "Operons," *Cellular and Molecular Life Sciences*, vol. 66, p. 3755—3775, Aug. 2009. doi: 10.1007/s00018-009-0114-3.

[8] "Global Interpreter Lock." [Online]. Available: https://docs.python.org/3/glossary.html#term-global-interpreter-lock. Accessed 21 Aug. 2024.

[9] "threading - Thread-based Parallelism." [Online]. Available: https://docs.python.org/3/library/threading.html. Accessed 21 Aug. 2024.

[10] "PEP 703 – Making the Global Interpreter Lock Optional in CPython." [Online]. Available: https://peps.python.org/pep-0703/. Accessed 14 Aug. 2024.

[11] Python Software Foundation, "Python 3.13.0rc1." [Online]. Available: https://www.python.org/downloads/release/python-3130rc1/. Accessed 14 Aug. 2024.

[12] Python Software Foundation, "What's New In Python 3.13." [Online]. Available: https://docs.python.org/3.13/whatsnew/3.13.html. Accessed 14 Aug. 2024.

[13] "MMseqs2 GitHub Repository." [Online]. Available: https://github.com/soedinglab/MMseqs2. Accessed 12 Aug. 2024.

[14] T. Frickey and A. Lupas, "CLANS: a Java application for visualizing protein families based on pairwise similarity," *Bioinformatics*, vol. 20, pp. 3702–3704, July 2004. doi: 10.1093/bioinformatics/bth444.

[15] Max Planck Institute for Biology, Tübingen, "CLANS." [Online]. Available: https://toolkit.tuebingen.mpg.de/tools/clans. Accessed 13 Aug. 2024.

[16] National Center for Biotechnology Information (NCBI), "FAQ: What is the difference between a GenBank (GCA) and RefSeq (GCF) genome assembly?." [Online]. Available: https://www.ncbi.nlm.nih.gov/datasets/docs/v2/troubleshooting/faq/. Accessed 11 Aug. 2024.

[17] "SciPy: Fundamental Algorithms for Scientific Computing in Python." [Online]. Available: https://scipy.org/. Accessed 12 Aug. 2024.

[18] NCBI: National Center for Biotechnology Information, "Basic Local Alignment Search Tool." [Online]. Available: https://blast.ncbi.nlm.nih.gov/Blast.cgi. Accessed 12 Aug. 2024.

[19] Stockholm Bioinformatics Center (SBC), "Phobius: A combined transmembrane topology and signal peptide predictor." [Online]. Available: https://phobius.sbc.su.se/. Accessed 13 Aug. 2024.

[20] Danmarks Tekniske Universitet (DTU) Health Tech, "TMHMM-2.0 (outdated)." [Online]. Available: https://services.healthtech.dtu.dk/services/TMHMM-2.0/. Accessed 20 Aug. 2024.

[21] Danmarks Tekniske Universitet (DTU) Health Tech, "DeepTMHMM-1.0." [Online]. Available: https://services.healthtech.dtu.dk/services/DeepTMHMM-1.0/. Accessed 20 Aug. 2024.

[22] National Center for Biotechnology Information (NCBI), "NCBI Insights." [Online]. Available: https://ncbiinsights.ncbi.nlm.nih.gov/2018/04/19/testing-periods-for-new-api-keys/. Accessed 17 Aug. 2024.

[23] C. K. Saha, R. Sanches Pires, H. Brolin, M. Delannoy, and G. C. Atkinson, "FlaGs and webFlaGs: discovering novel biology through the analysis of gene neighbourhood conservation," *Bioinformatics*, vol. 37, pp. 1312–1314, Dec. 2020. doi: 10.1093/bioinformatics/btaa788.

[24] K. Ernits, C. K. Saha, T. Brodiazhenko, B. Chouhan, A. Shenoy, J. A. Buttress, J. J. Duque-Pedraza, V. Bojar, J. A. Nakamoto, T. Kurata, A. A. Egorov, L. Shyrokova, M. J. O. Johansson, T. Mets, A. Rustamova, J. Džigurski, T. Tenson, A. Garcia-Pino, H. Strahl, A. Elofsson, V. Hauryliuk, and G. C. Atkinson, "The structural basis of hyperpromiscuity in a core combinatorial network of type II toxin–antitoxin and related phage defense systems," *Proceedings of the National Academy of Sciences*, vol. 120, Aug. 2023. doi: 10.1073/pnas.2305393120.

[25] J. Botas, Á. Rodríguez del Río, J. Giner-Lamia, and J. Huerta-Cepas, "GeCoViz: genomic context visualisation of prokaryotic genes from a functional and evolutionary perspective," *Nucleic Acids Research*, vol. 50, pp. W352–W357, May 2022. doi: 10.1093/nar/gkac367.

[26] L. Overmars, R. Kerkhoven, R. J. Siezen, and C. Francke, "MGcV: the microbial genomic context viewer for comparative genome analysis," *BMC Genomics*, vol. 14, art. no. 209, 2013. doi: 10.1186/1471-2164-14-209.

[27] A. M. Cleary and A. D. Farmer, "Genome Context Viewer (GCV) version 2: enhanced visual exploration of multiple annotated genomes," *Nucleic Acids Research*, vol. 51, pp. W225–W231, May 2023. doi: 10.1093/nar/gkad391.

[28] NumPy Team, "NumPy - The fundamental package for scientific computing with Python." [Online]. Available: https://numpy.org/. Accessed 28 Aug. 2024.

[29] Python Package Index, "Pandas." [Online]. Available: https://pypi.org/project/pandas/. Accessed 26 Aug. 2024.

[30] Anaconda, "Numba: A High Performance Python Compiler." [Online]. Available: https://numba.pydata.org/. Accessed 21 Aug. 2024.

[31] Python Software Foundation, "multiprocessing - Process-based Parallelism." [Online]. Available: https://docs.python.org/3/library/multiprocessing.html. Accessed 21 Aug. 2024.

[32] Python Software Foundation, "concurrent.futures - Launching Parallel Tasks." [Online]. Available: https://docs.python.org/3/library/concurrent.futures.html. Accessed 21 Aug. 2024.

[33] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," 2023. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf. Accessed 21 Aug. 2024.

[34] L. Dalcin and Y.-L. L. Fang, "mpi4py: Status Update After 12 Years of Development," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 47–54, 2021. doi: 10.1109/MCSE.2021.3083216.

[35] M. Rogowski, S. Aseeri, D. Keyes, and L. Dalcin, "mpi4py.futures: MPI-Based Asynchronous Task Execution for Python," *IEEE Transactions on Parallel & Distributed Systems*, vol. 34, pp. 611–622, Feb. 2023. doi: 10.1109/TPDS.2022.3225481.

[36] L. Dalcin, "mpi4py.futures." [Online]. Available: https://mpi4py.readthedocs.io/en/stable/mpi4py.futures.html. Accessed 15 Aug. 2024.

[37] M. Rocklin, "Dask: Parallel computation with Bllocked algorithms and Task Scheduling," in *Proceedings of the 14th Python in Science Conference*, pp. 126–132, 2015. doi: 10.25080/Majora-7b98e3ed-013.

[38] Anaconda Inc. and Contributors, "Dask: A Python library for parallel and distributed computing." [Online]. Available: https://docs.dask.org/en/stable/. Accessed 15 Aug. 2024.

[39] SchedMD, "SLURM Workload Manager." [Online]. Available: https://slurm.schedmd.com/documentation.html. Accessed 15 Aug. 2024.

[40] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "PyCOMPSs: Parallel computational workflows in Python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017. doi: 10.1177/1094342015594678.

[41] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive Parallel Programming in Python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, (New York, NY, USA), pp. 25–36, Association for Computing Machinery, 2019. doi: 10.1145/3307681.3325400.

[42] H. Lee, W. Ruys, I. Henriksen, A. Peters, Y. Yan, S. Stephens, B. You, H. Fingler, M. Burtscher, M. Gligoric, K. Schulz, K. Pingali, C. J. Rossbach, M. Erez, and G. Biros, "Parla: A Python Orchestration System for Heterogeneous Architectures," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2022. doi: 10.1109/SC41404.2022.00056.

[43] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. Scogland, B. Springmeyer, and M. Taufer, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Generation Computer Systems*, vol. 110, pp. 202–213, 2020. doi: 10.1016/j.future.2020.04.006.

[44] Flux Framework Community, "Flux: A flexible framework for resource management customized for your HPC site." [Online]. Available: https://flux-framework.org/. Accessed 21 Aug. 2024.

[45] Flux Framework Community, "Flux Python Bindings GitHub Repository." [Online]. Available: https://github.com/flux-framework/flux-python?tab=readme-ov-file. Accessed 21 Aug. 2024.

[46] A. Al-Saadi, D. H. Ahn, Y. Babuji, K. Chard, J. Corbett, M. Hategan, S. Herbein, S. Jha, D. Laney, A. Merzky, T. Munson, M. Salim, M. Titov, M. Turilli, T. D. Uram, and J. M. Wozniak, "ExaWorks: Workflows for Exascale," in *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pp. 50–57, 2021. doi: 10.1109/WORKS54523.2021.00012.

[47] ExaWorks, "PSI/J GitHub Repository: Portable Submission Interface for Jobs." [Online]. Available: https://github.com/ExaWorks/psij-python/tree/main. Accessed 21 Aug. 2024.

[48] ExaWorks, "Software Development Kit GitHub Repository." [Online]. Available: https://github.com/ExaWorks/SDK. Accessed 21 Aug. 2024.

[49] V. S. S. Peddinti, V. R. Mandla, S. Mesapam, and S. Kancherla, "Python parallel processing for hyperspectral image simulation: based on distance functions," *Earth Science Informatics*, vol. 14, pp. 2221–2229, Sep. 2021. doi: 10.1007/s12145-021-00690-7.

[50] F. Ye, L. Cui, Y. Zhang, Z. Wang, S. Moghimi, E. Myers, G. Seroka, A. Zundel, S. Mani, and J. G. Kelley, "A parallel Python-based tool for meshing watershed rivers at continental scale," *Environmental Modelling & Software*, vol. 166, art. no. 105731, 2023. doi: 10.1016/j.envsoft.2023.105731.

[51] M. Ledum, M. Carrer, S. Sen, X. Li, M. Cascella, and S. L. Bore, "HylleraasMD: Massively parallel hybrid particle-field molecular dynamics in Python," *Journal of Open Source Software*, vol. 8, no. 84, art. no 4149, 2023. doi: 10.21105/joss.04149.

[52] Andrew Collette & Contributors, "h5py: HDF5 for Python Documentation." [Online]. Available: https://docs.h5py.org/en/stable/. Accessed 28 Aug. 2024.

[53] L. Szustak, M. Lawenda, S. Arming, G. Bankhamer, C. Schweimer, and R. Elsässer, "Profiling and optimization of Python-based social sciences applications on HPC systems by means of task and data parallelism," *Future Generation Computer Systems*, vol. 148, pp. 623–635, 2023. doi: 10.1016/j.future.2023.07.005.

[54] D. G. A. Smith, D. Altarawy, L. A. Burns, M. Welborn, L. N. Naden, L. Ward, S. Ellis, B. P. Pritchard, and T. D. Crawford, "The MolSSI QCArchive project: An open-source platform to compute, organize, and share quantum chemistry data," *WIREs Computational Molecular Science*, vol. 11, no. 2, art. no. e1491, 2021. doi: 10.1002/wcms.1491.

[55] COMPSs, "PyCOMPSs + Jupyter Tutorial Notebooks." [Online]. Available: https://github.com/bsc-wdc/notebooks?tab=readme-ov-file. Accessed 16 Aug. 2024.

[56] Universal Protein Resource Database (UniProt), "ID mapping." [Online]. Available: https://www.uniprot.org/help/id_mapping. Accessed 17 Aug. 2024.

[57] CPython Lib , "Multiprocessing pool.py Implementation." [Online]. Available: https://github.com/python/cpython/blob/main/Lib/multiprocessing/pool.py. Accessed 28 Aug. 2024.

[58] Anaconda Inc. and Contributors, "Dask DataFrame." [Online]. Available: https://docs.dask.org/en/stable/dataframe.html. Accessed 17 Aug. 2024.

[59] Apache Parquet, "Parquet Documentation." [Online]. Available: https://parquet.apache.org/docs/. Accessed 28 Aug. 2024.

[60] Dask community forum, "Difference when starting a SLURM cluster with Conda from SLURM job or from Terminal." [Online]. Available: https://dask.discourse.group/t/difference-when-starting-a-slurm-cluster-with-conda-from-slurm-job-or-from-terminal/2967. Accessed 21 Aug. 2024.

[61] W. McGugan, "Rich." [Online]. Available: https://rich.readthedocs.io/en/stable/introduction.html. Accessed 18 Aug. 2024.

[62] Python Software Foundation, "logging — Logging facility for Python." [Online]. Available: https://docs.python.org/3/library/logging.html. Accessed 18 Aug. 2024.

[63] pyyaml.org, "Documentation for PyYAML and libyaml." [Online]. Available: https://github.com/yaml/pyyaml.org. Accessed 18 Aug. 2024.

[64] Anaconda, Inc., "Working with Conda: Environments." [Online]. Available: https://docs.anaconda.com/working-with-conda/environments/. Accessed 18 Aug. 2024.

[65] R. Krummenacher, "GCsnap: gcsnap2desktop." [Online]. Available: https://github.com/RetoKrummenacher/GCsnap/tree/gcsnap2desktop. Accessed 30 Aug. 2024.

[66] R. Krummenacher, "GCsnap: gcsnap2cluster." [Online]. Available: https://github.com/RetoKrummenacher/GCsnap/tree/gcsnap2cluster. Accessed 30 Aug. 2024.

[67] "pypdl: A concurrent pure python download manager." [Online]. Available: https://pypi.org/project/pypdl/. Accessed 18 Aug. 2024.

[68] Python Software Foundation, "urllib.request — Extensible library for opening URLs." [Online]. Available: https://docs.python.org/3/library/urllib.request.html. Accessed 18 Aug. 2024.

[69] Python Software Foundation, "sqlite3 — DB-API 2.0 interface for SQLite databases." [Online]. Available: https://docs.python.org/3/library/sqlite3.html. Accessed 20 Aug. 2024.

[70] NCBI: National Center for Biotechnology Information, "RefSeq Frequently Asked Questions (FAQ): How frequently are RefSeq records updated?." [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK50679/#RefSeqFAQ.how_frequently_are_refseq_reco. Accessed 27 Aug. 2024.

[71] Universal Protein Resource Database (UniProt), "How frequently is UniProt released?." [Online]. Available: https://www.uniprot.org/help/synchronization. Accessed 27 Aug. 2024.

[72] Forschungszentrum Jülich, "Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes." [Online]. Available: https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/index.html. Accessed 30 Aug. 2024.

Appendix A

GCsnap User Survey

GCsnap User Survey

We are conducting this survey to better understand how users interact with GCsnap, including usage patterns, integration into workflows, and areas where we can improve or enhance functionality to better meet your needs.

We do not collect your e-mail address. Your responses are anonymous.

This survey should take **no more than 15 minutes** to complete. Thank you for your time and insight.

---

* Indicates required question

General Usage

1. What have you used GCsnap for the most? *

   Mark only one oval.

   - Gene discovery and annotation
   - Comparative genomics
   - Functional genomics
   - Discover new functional relationships
   - Other: ____________________

2. How often do you use GCsnap? *

   Mark only one oval.

   - Daily: I use GCsnap every day.
   - Weekly: I use GCsnap at least once a week.
   - Monthly: I use GCsnap at least once a month.
   - Occasionally: I use GCsnap less frequently than once a month.
   - No longer using: I have used GCsnap in the past, but I no longer use it.

(a) Introduction and general usage.

Figure A.1: GCsnap user survey questions.

3. Do you use GCsnap as a stand-alone tool or as part of a larger genomic analysis workflow? *

Mark only one oval.

- ◯ Stand-alone
- ◯ Part of workflow

4. If part of a workflow, please describe how you typically use GCsnaps in your projects.

____

____

____

____

____

User Experience

5. How would you rate the ease of the installation process of GCsnap? *

Mark only one oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very easy

6. How helpful was the documentation provided on the GCsnap GitHub repository for the installation process? *
https://github.com/JoanaMPereira/GCsnap

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very helpful

(b) General usage (cont.) and user experience.

Figure A.1: GCsnap user survey questions (cont.).

7. How would you rate the ease of execution of GCsnap? *

Mark only one oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very easy

8. How helpful was the documentation provided on the GCsnap GitHub repository for the execution? *
https://github.com/JoanaMPereira/GCsnap

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very helpful

Features

9. Which feature of GCsnap do you find most useful? *

Mark only one oval.

- ◯ Interactive Output
- ◯ Customizable visualization options
- ◯ Ability to handle various input formats
- ◯ Combining multiple data sources (NCBI, UniProt, SwissModel)
- ◯ Other: ______________________

(c) User experience (cont.) and features.

Figure A.1: GCsnap user survey questions (cont.).

10. Are there any features that you find redundant?

11. Are there any features that you feel are missing from the tool?

Executing GCsnap

12. Under which operating system are you running GCsnap? *

Mark only one oval.

- Linux distribution (e.g. Ubuntu, Fedora)
- MacOS
- Windows
- Other: ______________

(d) Features (cont.) and execution experience.

Figure A.1: GCsnap user survey questions (cont.).

13. Which of the following optional argument categories do you regularly use when working with GCsnap? The arguments from each category are listed in parentheses. *

Please do not select more than the three that you use the most.

Check all that apply.

- [ ] Generall execution (get_taxonomy, collect_only)
- [ ] Parallel execution (cpu)
- [ ] Folder control (tmp_folder, out_label, out_label_suffix)
- [ ] Number of flanking sequences (n_flanking, n_flanking5, n_flanking3)
- [ ] Operon clustering (exclude_partial, n_max_operons, operon_cluster_advanced, max_family_freq, min_family_freq)
- [ ] Protein family identification (n_iterations, evalue, coverage, base, all-against-all_method, psiblast_location, mmseqs_location)
- [ ] Figure making (genomic_context_map, out_format, print_color_summary)
- [ ] Annotation (get_pdb, get_functional_annotations, annotate_TM, annotation_TM_mode, annotation_TM_file)
- [ ] Interactive output (interactive, gc_legend_mode, min_coocc, min_freq_accross_context, sort_mode, int_tree, in_tree_format)
- [ ] Clans map (clans_patterns, clans_file)

14. Did you know that GCsnap supports this many arguments? *

Mark only one oval.

- Yes
- No

15. Would you like to pass arguments through a configuration file? *

Mark only one oval.

- Yes
- No

(e) Execution experience (cont.).

Figure A.1: GCsnap user survey questions (cont.).

16. Have you ever executed GCsnap in parallel by specifying the *cpu* argument? This option sets the number of CPUs used to run GCsnap. *

Mark only one oval.

- Yes
- No
- I was not aware that this option existed.

17. How would you like to execute GCsnap in terms of parallelism? *

Mark only one oval.

- I don't need GCsnap to run in parallel. It works fine for my purposes with only one CPU.
- I would like to specify how many CPUs GCsnap uses during execution.
- I want GCsnap to use all resources of my machine by default.
- I want to execute GCsnap on a computing cluster.
- I want to execute GCsnap on cloud platforms (AWS, Azure)
- Other: ______________________

18. Why would you execute GCsnap in parallel? *

Mark only one oval.

- I would not execute GCsnap in parallel.
- Speed improvement when analyzing a small number of input sequences.
- Efficiency and timeliness for medium-sized input: Study up to a hundred sequences in a reasonable amount of time.
- Enable large-scale genomic studies: Analyze hundreds of sequences at once.
- Other: ______________________

(f) Execution experience (cont.).

Figure A.1: GCsnap user survey questions (cont.).

19. On average, how many input sequences (targets) do you use when running GCsnap? *

Mark only one oval.

- <5
- 5-9
- 10-19
- 20-50
- Other: ____________

20. What are the reasons why you do not use GCsnap with more input sequences at the same time? *

Mark only one oval.

- No specific reason.
- Performance: GCsnap takes too long to compute.
- The produced output with many sequences offers no insight anymore.
- Not needed as my analysis focuses on a smaller, more specific set of proteins.
- Other: ____________

21. Have you encountered any bugs or problems while using the GCsnap? Please describe them.

____

____

____

____

____

(g) Execution experience (cont.).

Figure A.1: GCsnap user survey questions (cont.).

Open Feedback

22. Any other comments or suggestions?

(h) General feedback.

Figure A.1: GCsnap user survey questions (cont.).

Appendix B

Additional Plots and Tables

B.1 GCsnap1 Execution Time Analysis

Table B.1: List of the 26 combinations of targets and CPU cores resulting in failed experiment when running GCsnap1.

Number of targets	CPU cores	Number of failed repetitions
10	8	1
20	16	2
20	64	1
50	4	1
100	16	1
200	1	1
200	4	1
200	32	4
200	64	3
500	1	4
500	4	1
500	8	1
500	16	1
500	32	5
500	64	4
1'000	1	5
1'000	16	1
1'000	32	5
1'000	64	5
2'000	1	2
2'000	2	1
2'000	4	2
2'000	8	2
2'000	16	3
2'000	32	5
2'000	64	5

Figure B.1: Average end-to-end execution time of GCsnap1 over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished (see Appendix B.1).

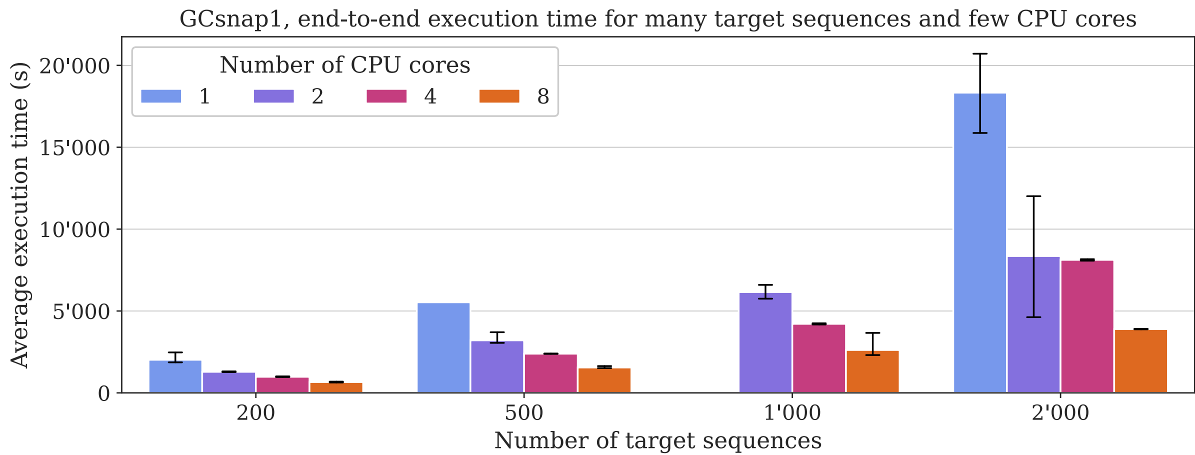


Figure B.2: Average end-to-end execution time of GCsnap1 over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished (see Appendix B.1).

Figure B.3: Average end-to-end execution time of GCsnap1 over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished (see Appendix B.1).

B.2 ID Mapping with Dask.DataFrame

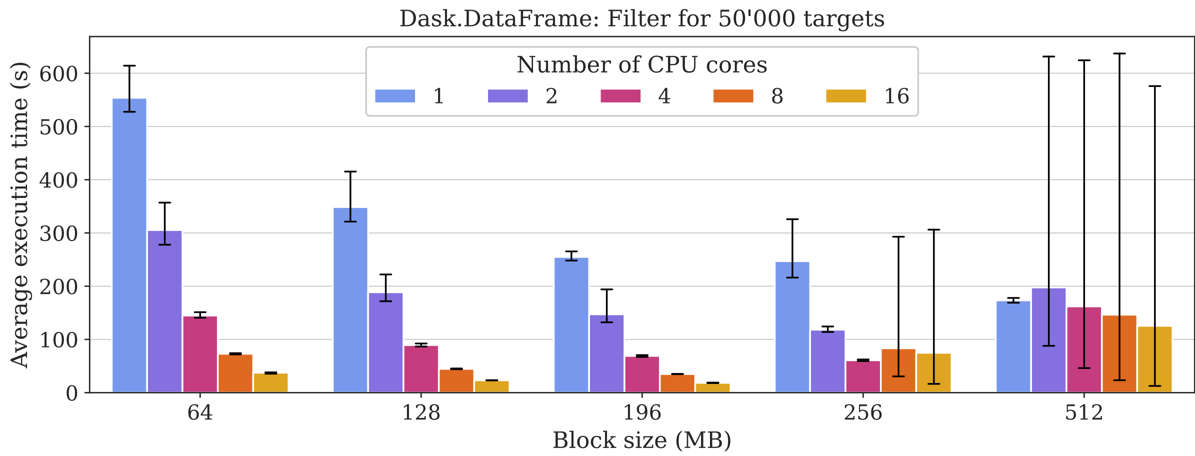


Figure B.4: Average execution time of the ID mapping experiment with Dask.DataFrame over 5 repetitions with different numbers of CPU cores and block sizes with 50’000 targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on 2 Xeon E5-2640v4 with a total of 20 CPU cores.

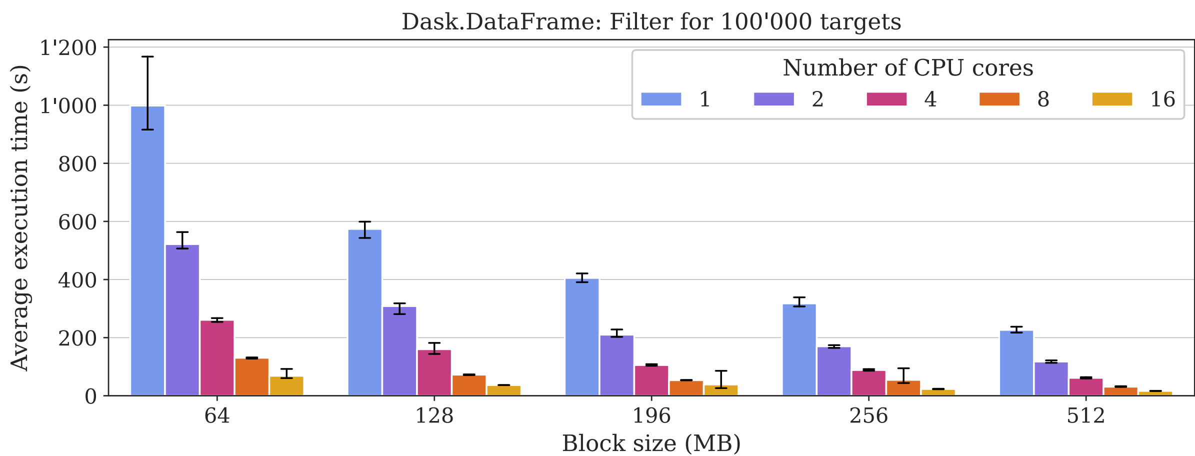


Figure B.5: Average execution time of the ID mapping experiment with Dask.DataFrame over 5 repetitions with different numbers of CPU cores and block sizes with 100’000 targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on 2 Xeon E5-2640v4 with a total of 20 CPU cores.

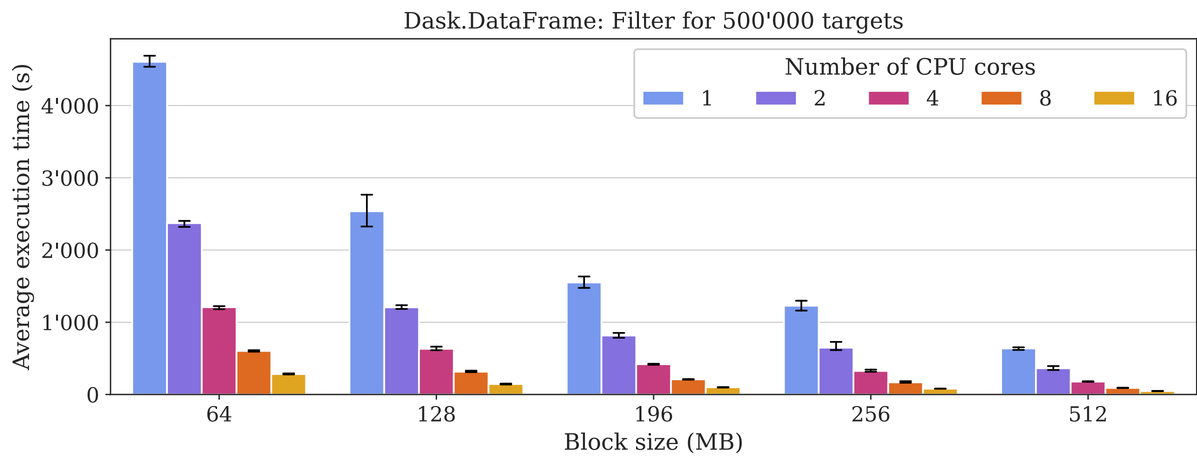


Figure B.6: Average execution time of the ID mapping experiment with Dask.DataFrame over 5 repetitions with different numbers of CPU cores and block sizes with 500'000 targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on 2 Xeon E5-2640v4 with a total of 20 CPU cores.

B.3 Assembly parsing with Dask.distributed and mpi4py.futures

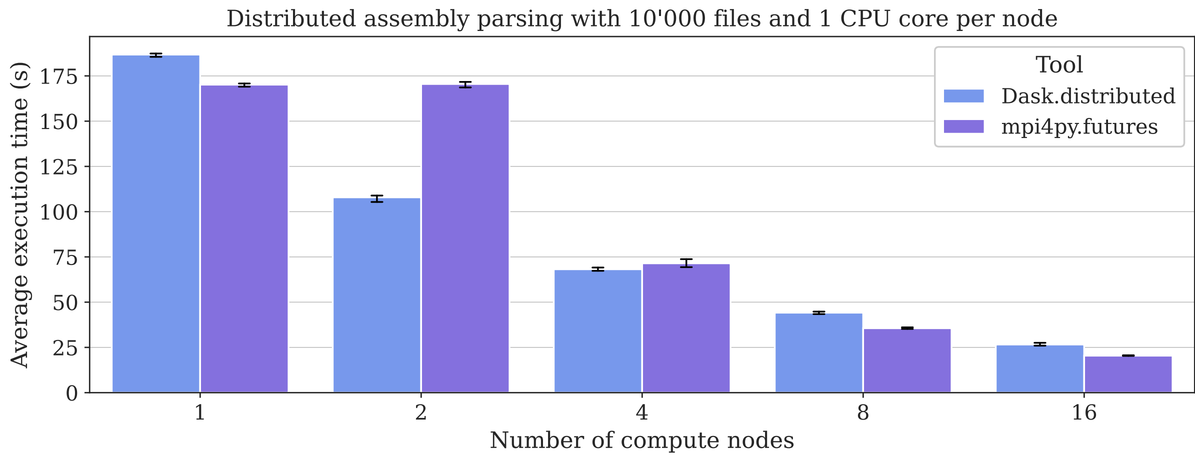


Figure B.7: Average execution time of the assembly file parsing experiment with Dask.distributed and mpi4py.futures over 5 repetitions with different numbers of nodes, 1 CPU core per node, and 10'000 files. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on miniHPC nodes, each with 2 Xeon E5-2640v4 and a total of 20 CPU cores.

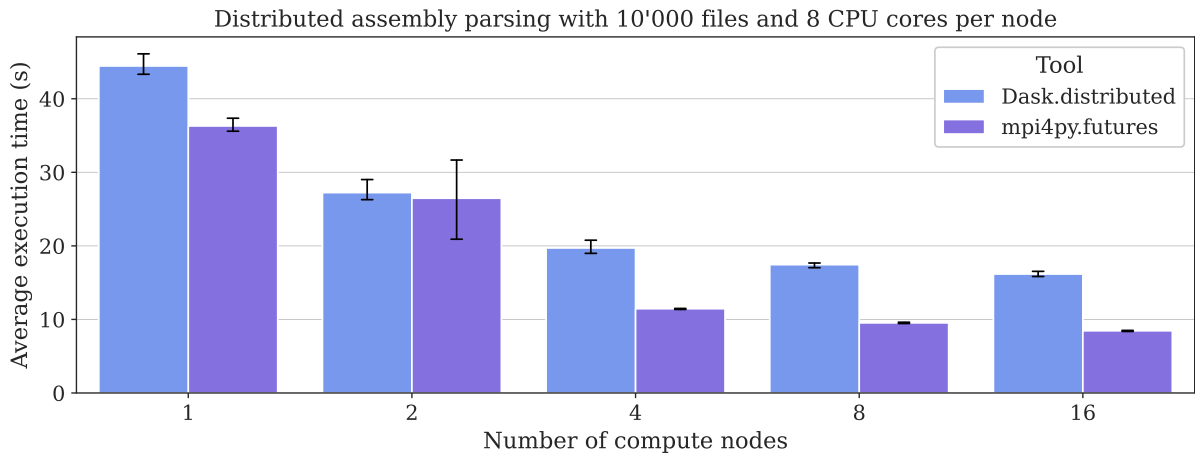


Figure B.8: Average execution time of the assembly file parsing experiment with Dask.distributed and mpi4py.futures over 5 repetitions with different numbers of nodes, 8 CPU cores per node, and 10'000 files. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on miniHPC nodes, each with 2 Xeon E5-2640v4 and a total of 20 CPU cores.

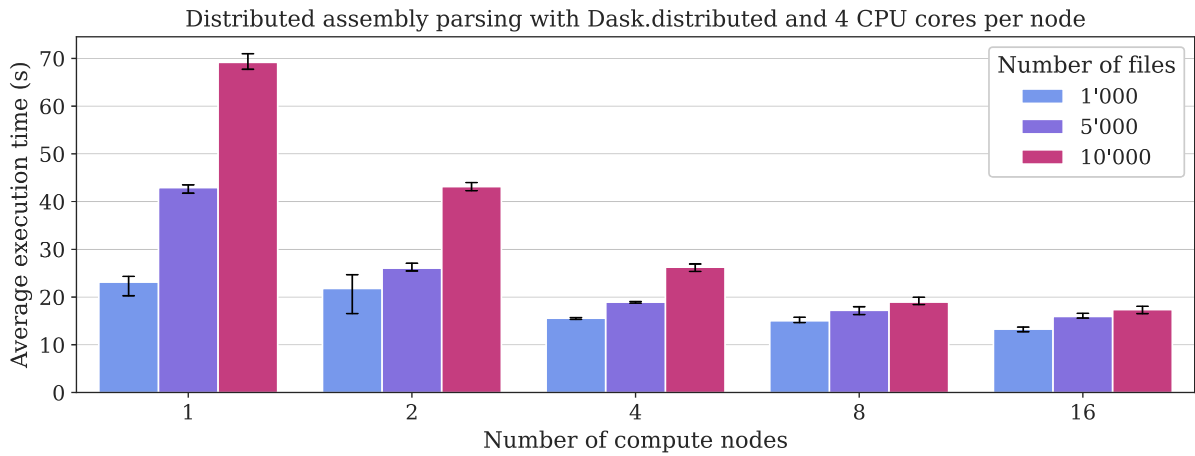


Figure B.9: Average execution time of the assembly file parsing experiment with Dask.distributed over 5 repetitions with different numbers of files and 4 CPU core per node. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on miniHPC nodes, each with 2 Xeon E5-2640v4 and a total of 20 CPU cores.

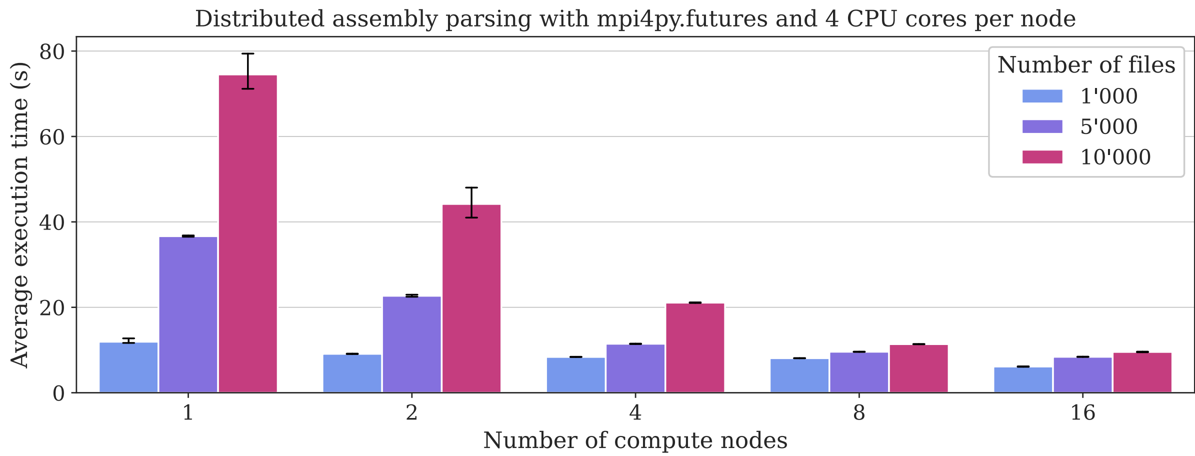


Figure B.10: Average execution time of the assembly file parsing experiment with mpi4py.futures distributed over 5 repetitions with different numbers of files and 4 CPU cores per node. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on miniHPC nodes, each with 2 Xeon E5-2640v4 and a total of 20 CPU cores.

B.4 GCsnap2.0 Desktop Performance Analysis

Table B.2: List of the 9 combinations of targets and CPU cores resulting in failed experiment when running GCsnap2.0 Desktop.

Number of targets	CPU cores	Number of failed repetitions
500	32	1
1'000	1	3
1'000	16	1
1'000	64	1
2'000	1	5
2'000	2	4
2'000	8	2
2'000	16	1

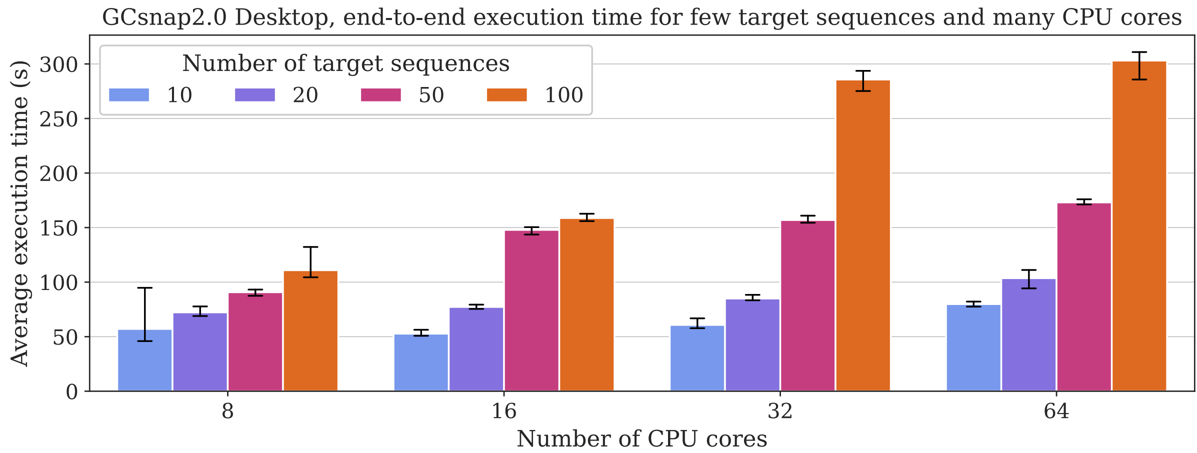


Figure B.11: Average end-to-end execution time of GCsnap2.0 Desktop over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished (see Appendix B.4).

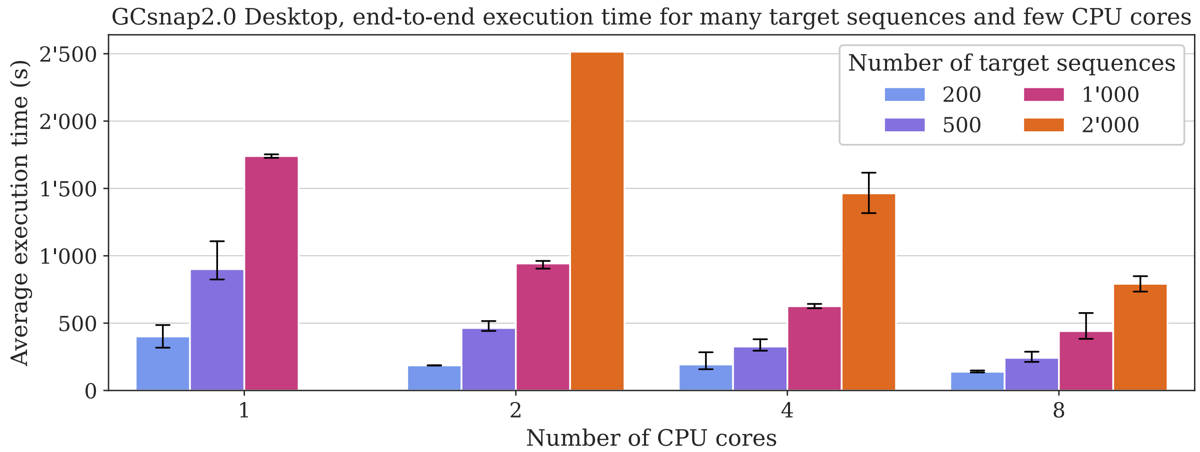


Figure B.12: Average end-to-end execution time of GCsnap2.0 Desktop over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished (see Appendix B.4).

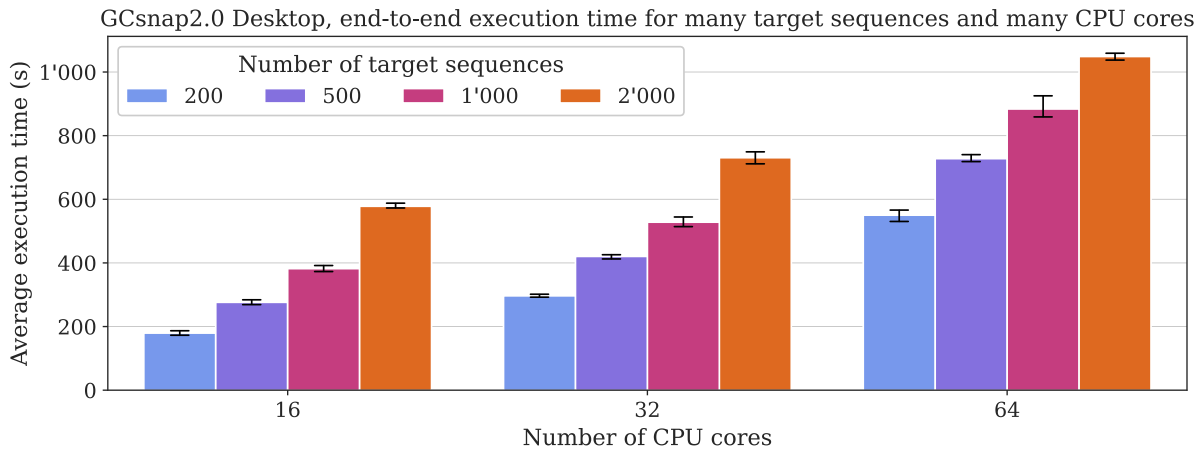


Figure B.13: Average end-to-end execution time of GCsnap2.0 Desktop over 5 repetitions with different numbers of CPU cores and input targets. Error bars represent the minimum and maximum over the 5 repetitions. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished (see Appendix B.4).

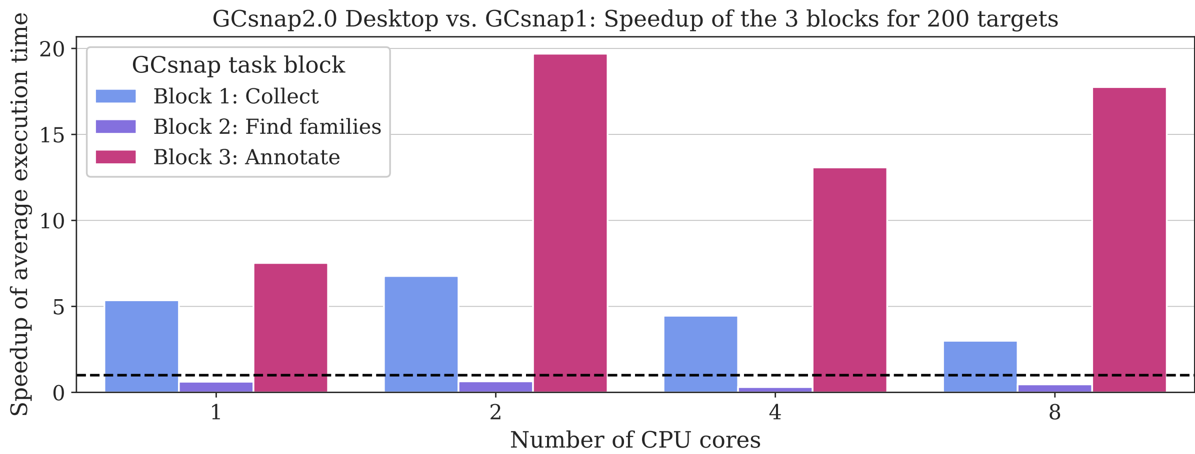


Figure B.14: Speedup of the average end-to-end execution time over 5 repetitions of gc2D compared to GCsnap1 with different numbers of CPU cores and 200 targets for the 3 blocks of. A value above 1 (black dashed line) means that gc2D had a lower average execution time. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished (see Appendix B.1 and Appendix B.4)

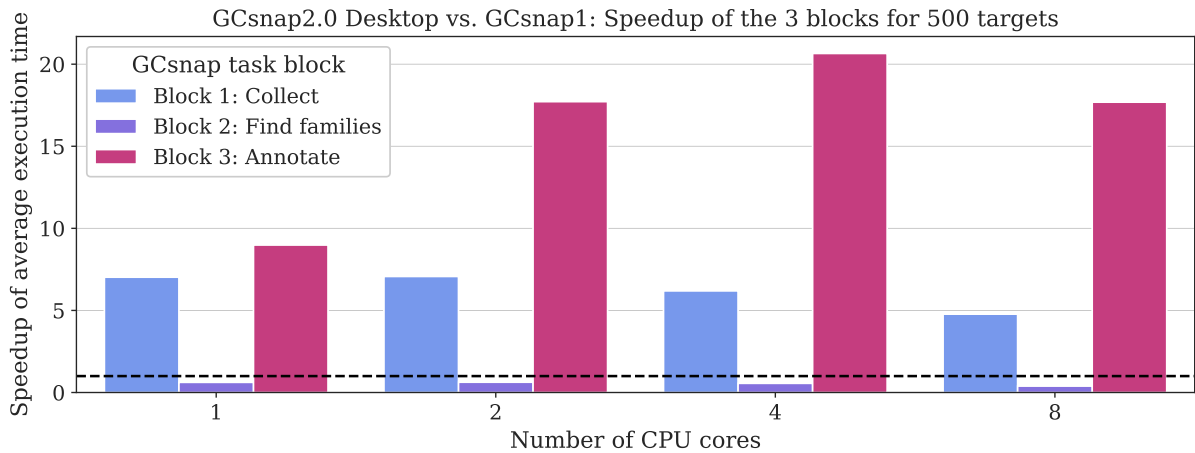
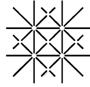


Figure B.15: Speedup of the average end-to-end execution time over 5 repetitions of gc2D compared to GCsnap1 with different numbers of CPU cores and 500 targets for the 3 blocks of. A value above 1 (black dashed line) means that gc2D had a lower average execution time. Experiments were conducted on an AMD EPYC 7742 with 64 CPU cores. Not all repetitions finished (see Appendix B.1 and Appendix B.4)



University of Basel

Faculty of Science

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: A Million Proteins per Second: Scaling a Protein Comparison Tool for Biological Breakthroughs

Name Assessor: Prof. Dr. Florina M. Ciorba

Name Student: Reto Krummenacher

Matriculation No.: 03-054-327

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: Basel, 30.8.24 Student: [signature]

Will this work, or parts of it, be published?

☐ No

☑ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: ____________________

Place, Date: Basel, 30.8.24 Student: [signature]

Place, Date: ____________________ Assessor: ____________________

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis.

September 2023