Large Scale Parallel Inference of Protein and Protein Domain families

Clément Rezvoy

N° d'ordre : — *N° attribué par la bibliothèque : ———*

- École normale supérieure de Lyon -
Laboratoire de l'Informatique du Parallélisme

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon - École normale supérieure de Lyon
Spécialité : Informatique

au titre de l'École Doctorale InfoMaths

présentée et soutenue publiquement le 28 Septembre 2011 par

Clément REZVOY

Large Scale Parallel Inference of Protein and Protein Domain Families

Directeur de thèse :	Frédéric	VIVIEN	
Co-directeur de thèse :	Daniel	KAHN	
Après avis de :	Dominique	LAVENIER	Rapporteur
	Marco	PAGNI	Rapporteur

Devant la commission d'examen formée de :

Gilbert	DELÉAGE	Membre
Daniel	KAHN	Membre
Dominique	LAVENIER	Rapporteur
Johan	MONTAGNAT	Membre
Marco	PAGNI	Rapporteur
Frédéric	VIVIEN	Membre

Résumé

Les progrès des techniques de séquençage au cours de la dernière décennie ont permis aux chercheurs en sciences de la vie d'avoir accès à une quantité toujours plus importante de données génomiques. Si ces nouvelles données permettent des avancées importantes en biologie, médecine ou encore agronomie, elle nécessitent également de développer de nouvelles méthodes d'analyse capable de traiter la quantité toujours grandissante de données issues des projets de séquençage de nouvelle génération.

Dans cette thèse, nous nous intéressons au problème de la classification des domaines protéiques en familles de domaines apparentés. Un domaine protéique est une sous-partie d'une protéine qui fonctionne et évolue de façon indépendante du reste de la protéine. Des domaines apparentés peuvent être présents dans différentes protéines et l'arrangement combinatoire de ces domaines est à l'origine de la diversité structurale et fonctionnelle des protéines. Plusieurs méthodes ont été développées pour permettre l'identification de ces domaines au sein des séquences de protéines et permettre leur classification en familles de domaines apparentés de façon automatique. Nous étudions le cas de l'une de ces méthodes, MKDOM2. MKDOM2 est un algorithme glouton séquentiel : à partir d'un jeu de séquences protéiques, MKDOM2 infère les familles de domaines les unes après les autres de façon à créer une partition de l'ensemble des séquences protéiques en familles de domaines non-recouvrantes. MKDOM2 est un élément essentiel de la construction de ProDom, une base de données de familles de domaines protéiques. Aujourd'hui, la complexité quadratique de MKDOM2 et sa séquentialité ont rendu impossible la mise à jour de ProDom dans des délais raisonnables. Il faudrait actuellement plus de 10 ans pour pouvoir calculer une nouvelle version de la base de données à partir de l'ensemble des données disponibles.

Afin de permettre la mise à jour de ProDom, il est nécessaire de définir une nouvelle méthode permettant d'inférer efficacement les familles de domaines protéiques tout en préservant la qualité des résultats que fournit MKDOM2. Nous proposons un nouvel algorithme distribué, MPI_MKDOM2, qui, tout en préservant les principes de l'heuristique que MKDOM2, permet le calcul simultané de plusieurs familles sur une plate-forme de calcul distribué. MPI_MKDOM2 doit s'assurer que bien que plusieurs familles sont définies en même temps, le résultat final ne contient pas de familles chevauchantes. MPI_MKDOM2 essaye, par le biais d'un mécanisme de prédiction, d'éviter au maximum de calculer simultanément des familles pouvant se chevaucher. Les familles conflictuelles pour lesquelles cette prédiction échoue sont détectées et recalculées suivant la logique séquentielle. Tout en permettant la parallélisation du calcul des familles, MPI_MKDOM2 garantit que le résultat final est bien composé de familles non-recouvrantes. Afin de d'assurer une utilisation efficace des ressources, MPI_MKDOM2 évite la synchronisation entre les calculs intermédiaires et ainsi évite les temps d'attente qui pénaliseraient les performances parallèles de l'algorithme. MPI_MKDOM2 à permis de créer une version mise à jour de ProDom basée sur l'ensemble des données disponible en 2010, calculant

en 20 jours ce qui aurait pris plus de 12 ans à calculer séquentiellement.

Une mesure de similarité entre les classifications de domaines protéiques a été définies afin d'étudier les effets de parallélisation sur les résultats. Cette mesure prend en compte à la fois les différences dans la définitions des domaines et les différences dans la classification des domaines en familles. Tout en réduisant de façon importante le temps de calcul nécessaire à la construction de ProDom, MPI_MKDOM2 crée des classifications qui restent cohérentes avec celles créées par l'algorithme séquentiel MKDOM2. Une classification en familles de domaines protéiques est souvent utilisée en conjonction avec une classification en familles de séquences protéiques pour des études phylogénétiques en génomique évolutive. Le calcul séparé de ces deux classifications entraîne des difficultés méthodologiques et des incohérences entre les deux classifications. Une solution possible est la construction de familles de domaines sur la base de familles de séquences pré-calculées, ce que nous avons réalisé dans un second algorithme, MPI_MKDOM3. Sur la base du travail de parallélisation effectué pour MPI_MKDOM2, ce nouvel algorithme permet la création coordonnée d'une classification de familles de domaines protéiques et d'une classification de familles de séquences protéiques constituées des mêmes domaines. Tout en améliorant le résultat des deux classifications, MPI_MKDOM3 permet également, en basant la construction des familles de domaines sur des familles de séquences et non plus sur les séquences elles-mêmes, une compression des données d'entrée qui vient atténuer l'augmentation exponentielle du nombre de séquences à traiter.

Abstract

In recent years, vast amounts of genomic data were made available through 'next generation' sequencing technologies. While this abundance of information drives new insight into many fields of life sciences, analysing these data also calls for the development of new methods that can handle the ever increasing load.

In this dissertation, we focus on the problem of protein domain clustering. Protein domains are discrete structural/evolutionary units that may occur in different proteins. The combinatorial arrangement of these domains is at the root of structural and functional protein diversity. Over the years, several methods have been developed to automatically infer the decomposition of proteins into domain arrangements, and the clustering of these domains into families. One such method is MKDOM2. MKDOM2 is a sequential algorithm that works in a greedy fashion: starting from a database of protein sequences, families are inferred one at a time in order to create a complete clustering of the protein space into a set of non-overlapping protein domain families. MKDOM2 has been instrumental in the building of ProDom, a protein domain family database. Because of its sequential nature and quadratic complexity, MKDOM2 has become unable to process the ever increasing number of protein sequences available.

In order to allow further updates of ProDom, a new method is needed that can efficiently infer protein domain families while preserving the quality of the results provided by MkDom2. We propose a new distributed algorithm, MPI_MKDOM2. While retaining the same basic heuristic, this algorithm differs from MKDOM2 in that it allows the exploration of several domain families in parallel across a distributed computing platform. MPI_MKDOM2 needs to ensure that, while multiple families are computed in parallel, the end result is still composed of non-overlapping domain families. MPI_MKDOM2 uses a prediction mechanism to avoid such overlaps as much as possible. Remaining overlapping results are detected and re-computed to ensure that no overlap remains in the final clustering. Problems of load-balancing and communication bottlenecks have been addressed. While MPI_MKDOM2 is able to achieve a reasonable speedup on a small database, the larger the database, the larger the achievable speedup. MPI_MKDOM2 has been successfully used to create a new version of ProDom based on all the complete protein sequences available as of 2010, achieving in 20 days a computation that would have taken more than 12 years to complete sequentially.

The impact of parallelization on clustering has been assessed by defining a measure of similarity between domain clusterings. This measure takes into account differences in both domain boundaries and domain clustering. While allowing to substantially decrease the construction time of ProDom, the new algorithm still creates results that are consistent with those of the original sequential algorithm.

Protein domain clustering is frequently used in conjunction with protein sequence clustering for phylogenetic studies in evolutionary genomics. However computing separately the two types of clusterings may lead to methodological difficulties when the two clusterings are inconsistent.

One possible solution is to construct domain families from precomputed protein families, which we achieved with a second algorithm, MPI_MKDOM3. Building upon the parallel framework developed in MPI_MKDOM2, this new algorithm allows the creation of a coordinated clustering of protein domain families along with families of protein sequences having the same domain decomposition. While improving the results of both classifications, MPI_MKDOM3 also allows significant computing gains by basing the domain classification on sequence families rather than on individual sequences, resulting in compression of the input data in the face of the exponential increase of the number of sequences.

Contents

List of Figures

List of Algorithms

List of Tables

Introduction

Proteins are the products of genes and the building blocks of the metabolism of living organisms. A predominant challenge in molecular biology is therefore to understand how proteins function and how they have evolved. Proteins are often composed of independent structural and functional units called domains. Evolutionarily related domains can be found in different proteins. In fact, much of the functional and structural diversity of proteins arises from the combinatorial rearrangement of these domains across proteins. Understanding domain arrangements of proteins is therefore crucial for the interpretation of protein sequence data. Resolving protein domain arrangements is also a prerequisite for protein classification, either for bioinformatics purposes or for evolutionary studies.

Protein domains sharing a common evolutionary history can often be recognized by sequence similarity. Several methods have been proposed to identify and classify domains following this principle. Some databases gather domain families compiled by manually driven computational tools. Consequently, domain family definitions found in these databases rest on human expertise and experimental results such as structural information. These databases are therefore regarded as closely matching the "biological truth". The construction of these databases, however, requires expertise and experimental data which render them unsuitable for the comprehensive processing of the vast amount of protein sequence data available nowadays. Furthermore, the gap between these domain family databases and the raw databases of protein sequences is constantly widening as the size of the latter is increasing at an exponential pace. To bridge this gap, some bioinformaticians have thus designed systems to fully automatically build domain family databases.

One of these fully automated projects is the ProDom database. ProDom aims at being a comprehensive repository of families of homologous domains built by an automatic process based solely on sequence similarity. ProDom is built from all known protein sequences available and, since 1998, it has been successfully built using an algorithm called MKDOM2. This algorithm, however, is inherently a sequential and iterative algorithm, whose complexity is quadratic in the size of the protein database processed. Therefore, the exponential increase in processing speed over the years is not sufficient to maintain the construction time of ProDom at a reasonable level. Indeed, while MKDOM2 took 2 months to build a new version of ProDom in 2002, it needed more than 15 months in 2007. The latest release of ProDom was released in 2008, and it took one and a half year to build.

It has thus become impossible to produce new versions of ProDom using the sequential MKDOM2 program. In principle it could be possible to run incremental updates by clustering homologous domains into pre-existing ProDom families, and running MKDOM2 on the remaining sequences, which would limit the computational cost. However such a greedy incremental procedure would forbid the re-examination of previously inferred domain families, even in the light of contradictory new evidence. In order to avoid the latest release of ProDom to be the last,

we decided to design a new algorithm allowing the inference of protein domain family clustering at large scale on distributed computing platforms. This document relates our work on this topic.

Dissertation organization

This work is divided in an introductory chapter and 3 chapters outlining our contributions.

In Chapter 1, we give the information and related work required to understand the problem addressed in this dissertation. This preliminary chapter is split in two main parts. To begin with, we give an overview of the field of research, and the context of the problem from a biology and bioinformatics point of view. In a second part we focus on computational notions, tools, and platforms used throughout this work.

In chapter 2 we present the design, implementation and testing of MPI_MKDOM2, a parallel, distributed, and scalable algorithm allowing the computation of new ProDom releases on gigabit scale datasets. The design decisions are motivated, and the parallel performance and the scalability are tested on current scale dataset.

Our objective to create a distributed algorithm that closely reproduce the result of a sequential computation leads us in Chapter 3 to address the problem of the comparison of protein domain clusterings. Several criteria have been proposed in the literature to compare and assess differences between clusterings. We present some of these methods and our criteria to take into account both differences in domain delineation and domain clustering. Using these criteria, we then assess the effect of the parallelization on the clusterings produced by MPI_MKDOM2. The exponential growth of the input data has been known to cause the fragmentation of families of ProDom. We explain the cause of this fragmentation and assess its impact on the stability of the clusterings with respect to the size of the input data. We then investigate the use of an alternative methods to define families to improve the stability of the clusterings produced by ProDom.

In Chapter 4, we present a new algorithm, MPI_MKDOM3, which computes protein domain clusterings on families of globally homologous sequences. By grouping sequences into families, we aim at solving two problems at once. While we show in Chapter 2 that MPI_MKDOM2 is a highly scalable method capable of efficiently processing large scale datasets, it must use more and more resources to handle the growth of the input data. Grouping sequences into families would help contain the increase of the input data, and reduce the need for additional computing resources for each new release of ProDom. This new algorithm also allows the construction of a coherent classification of protein domain families, and families of protein sequences sharing the same decomposition into domains. We place our work in the context of the HOGENOM database of protein families. We present how the methods used for the construction of HOGENOM and ProDom can be chained to create a new pipeline of methods for building paired databases of protein domains and protein sequence families. We present the modification that must be applied in order to allow MPI_MKDOM2 to handle protein families instead of protein sequences. To motivate our choices, we present several methods to analyze homology between protein families and outline their respective merits and drawbacks in terms of accuracy of results and computational efficiency. As we did for MPI_MKDOM2, we demonstrate that this algorithm is applicable at large scale on a current exhaustive dataset of protein sequences. Finally we summarize our work and outline perspectives for further evolution of the methods.

Chapter 1

Preliminaries

In this chapter, we give the information and related work required to understand the problem addressed in this dissertation. The first three sections give an overview of the fields of research from a biology and bioinformatics point of view, the rest of the chapter presents computer science notions used throughout the rest of the manuscript. In Section 1.1, we give an introduction to the biological context of the problem, namely the study of protein sequences and protein domains and their evolutionary relationships. Biological sequence analysis makes great use of computational tools to detect similarities between sequences. In Section 1.2, we outline how these similarities are detected and present BLAST, PSI-BLAST and HMMER, three of the most widely used sequence similarity search tools. Section 1.3 presents various biological sequences databases. The various databases differ on the information they contain, from only protein sequences to highly curated and annotated data and in the way they are constructed, which ranges from manual verification to fully automatic inference.

Automatically constructing these databases and applying sequence similarity search at large scale requires massive amount of computation that can only be solved in a reasonable time frame through parallel and distributed computing. In Section 1.4 we present the parallel platforms used throughout this work. Finally in Section 1.5 we give a brief overview of MPI, the most widely used distributed application framework in high performance computing and the one that was used in our implementations.

1.1 Proteins and protein domains

Proteins are the products of genes and they are the gears of the metabolism of living organisms. A protein is a molecule composed of *amino acids* chained together in a sequence. Since amino acids lose their amine and acid functions when chained together, we use the term *amino acid residue* or simply *residue* to name what is left of an amino acid within a protein chain. Protein molecules fold into three-dimensional structures depending mainly on their sequences [3]. The spatial conformation of the folded protein, the relative position of the different residues, their exposure or not on the outside surface of the molecule, determine the functional properties of the protein. A protein can therefore be defined by three interdependent characteristics:

- its sequence in residues;
- its three dimensional structure once folded;
- its functions or capabilities in interacting with other molecules.

Protein sequences observed today have evolved from ancestral sequences through mutations of their coding genes and natural selection, a trial and error process compared by Jacob to the work of a tinkerer [26]. This "tinkering" leaves traces. Despite these mutations, two related or *homologous* sequences will in general bear a resemblance to their common ancestor and may therefore share some similarities themselves. Significant similarity between two sequences is interpreted as a sign of common origin or *homology*.

Some proteins can be viewed as a collage or concatenation of independent regions called *modules* or *domains*. Although belonging to a single amino acid chain, encoded by a single gene, these domains may have independent characteristics to some extent: they may fold [55], function and have evolved in part independently [10] from the rest of the protein. Homologous domains can be found in globally non-homologous proteins, often adopting the same structure and performing the same function. The creation of multi-domain proteins by a combinatorial arrangement of conserved blocks through evolution has been called domain shuffling [6] and is a major cause of protein diversity; it is estimated that multi-domain proteins make up about 80% of eukaryotic proteins and about 65% of prokaryotic proteins [4].

Proteins can be grouped according to their sequences, structures, or functions. Characterizing the functions or the structure of a protein is a difficult task requiring expertise and experimentation. Obtaining protein sequences however is nowadays comparatively easy and affordable. High throughput sequencing techniques have made available a wealth of genomic data, and protein sequences can be derived from their corresponding genes. From homology relationship between protein sequences one can define families of closely related proteins. If proteins from the same family have similar sequences, it is likely that they also have similar structures and possibly similar functions. Knowledge about the structure or function of a protein in a family may be transferred to other proteins by homology. Similarly to full proteins, domains can be grouped into families of closely related elements, based on their sequence or structural similarities.

1.2 Homology search algorithms

1.2.1 Protein alignment

From a purely computational point of view, a protein sequence can be view as a text string defined over a 20 letter alphabet, one for each possible residue:

$$\Sigma_{\mathcal{AA}} = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$$

Detecting similarities between two sequences in its simplest form consists in putting matching positions in correspondence between the two text strings, in a process called an alignment.

Figure 1.1 illustrates the alignment between two sequences. Exact match between positions are marked with the symbol '|'. Two diverging sequences may differ in two ways: new residues may be inserted in one of the sequences, or deleted in the other, creating *gaps* in the alignments, symbolized by the symbol '-' in our example; otherwise, residues can be substituted by others such as in the four leftmost positions in our example. Natural selection acts as a filter and some mutations, between chemically close residues are more frequently observed than others. In our example, substitutions between similar residues are marked by the symbol '+'.

The quality of an alignment is assessed by assigning a score to each match, substitution, and gaps. The most common scoring scheme for protein sequences is the BLOSUM62 matrix [24] represented in table 1.2.1. BLOSUM stands for BLOcks of amino acid SUbstitution Matrix.

```
NEPELQAHAGDCGVAKKAVALTNAVADFMPNANSAGAHALKK
   +|           |  ||  |    |  |||++|+  +||
GSGDLLYLDSLTFVGKKGV---NTIADYLPSCDSALYLDDLT
```

Figure 1.1: Example of an alignment between two sequences.

Match scores in the BLOSUM62 matrix were derived from observed substitutions in alignment of conserved blocks. Diagonal scores reflect the relative abundance of the different amino-acids. For instance, Tryptophan (`W`) is relatively rare compared to other residues, matching two Tryptophan residues is therefore more significant than matching more frequent amino-acids such as Leucine (`L`) or Serine (`S`). Additionally, the matrix contains score for 2 additional pseudo-residues: the `X` stands for "any residue"; the `*` stands for end of translation, it corresponds to the end of the gene coding for the protein. Gaps are usually scored negatively following an affine function $f(g) = G_o + (g-1)G_e$, where g is the length of the gap, G_o is the cost for opening a gap and G_e is the cost for extending the gap. By default in BLAST (see Section 1.2.2) algorithm for instance, $G_o = 11$ and $G_e = 1$.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	X	*
A	4																					
R	-1	5																				
N	-2	0	6																			
D	-2	-2	1	6																		
C	0	-3	-3	-3	9																	
Q	-1	1	0	0	-3	5																
E	-1	0	0	2	-4	2	5															
G	0	-2	0	-1	-3	-2	-2	6														
H	-2	0	1	-1	-3	0	0	-2	8													
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4												
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4											
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5										
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5									
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6								
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7							
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4						
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5					
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11				
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7			
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4		
X	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	
*	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	1

Table 1.1: BLOSUM62 scoring matrix.

The best alignment between two sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ can be found using the Needleman-Wunsch algorithm [39]. $\mathcal{S}_1$ and $\mathcal{S}_2$ may not be globally homologous and yet share a local homology such as for instance two proteins sharing a related domain. In that case, a global alignment of $\mathcal{S}_1$ and $\mathcal{S}_2$ would not get a good score while an alignment of a restriction of the two sequences to their homologous regions would be scored more favorably. The Smith-Waterman algorithm [45], a variation of the Needleman-Wunsch algorithm, allows to find the best local alignment between two sequences, that is, the best match between two subsequences of $\mathcal{S}_1$ and $\mathcal{S}_2$. Both algorithms use dynamic programming and have a time complexity of $\mathcal{O}(mn)$ where m is the length of $\mathcal{S}_1$ and n is the length of $\mathcal{S}_2$.

1.2.2 Blast

Several bioinformatic problems require aligning a query sequence against a larger set of target sequences to find local matches. For instance, one may ask if a protein of known function is present in the proteins produced by a recently sequenced genome yet to be annotated. Using the Smith-Waterman algorithm to answer this question would require trying all the possible local alignments on all the target sequences in order to usually find none or very few good alignments.

BLAST [1] stands for Basic Local Alignment Search Tool. BLAST is designed to efficiently search for local alignments between a query sequence and a database of target sequences. While the worst case complexity of BLAST is the same as the Smith-Waterman algorithm, BLAST heuristically avoids most of the unnecessary alignments by starting the alignments in high scoring regions and stopping when the alignment score drops below a certain threshold. A principle of a BLAST search can be decomposed in 3 steps:

1. All the k letter words, or *seeds*, that can be created from continuous portions of the query are listed. Usually, $k = 3$ for protein alignments.

2. Sequences in the target database are scanned for exact or nearly exact matches of the seeds.

3. If a seed matches, BLAST will try to *extend* the match on the left and right sides of the seed using a dynamic programming approach similar to that of the Smith-Waterman algorithm. The alignment score is allowed to drop locally to allow local variations between the two sequences. If the score drops more than a certain threshold, the alignment stops and the end of the alignment is defined as the position where the score was maximal.

Local alignments found by BLAST are often referred to as *matches*. Figure 1.2 illustrates a BLAST match between two sequences. The top sequence is the query sequence and the bottom one is a target sequence, the line in between gives the alignment score between the positions on the query and on the target sequence. The seed `ADF` from the query sequence finds a high scoring match on a target sequence (dark gray zone). The match with the seed is extended leftward and rightward from the seed. The score of the extension is defined as the sum of the scores from the start of the seed extension. Extension stops when the score drops by more than a certain threshold. In our example we set the tolerated drop to 15 points, so that the leftward alignment reaches its maximal score on position 14. BLAST will try to further extend the match to the left but stops on position 6, as the score drops by 17 points. The end of the alignment is defined as the last position where the maximum was reached. In our example, the final alignment is delineated by the light gray zone.

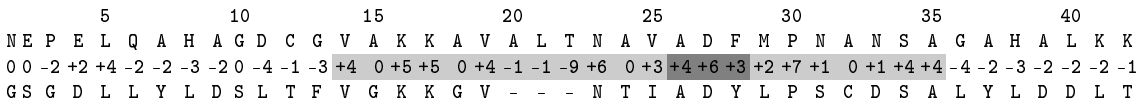

Figure 1.2: Illustration of an alignment with BLAST.

Estimation of the statistical significance of a match

Scoring schemes allow us to find the optimal alignment between two sequences. The score of an alignment does not however tell us if the alignment is biologically meaningful and reflects an

underlying homology, or if the two aligned sequences are completely unrelated. The statistical significance of a BLAST alignment is assessed through its *E-value* [28]:

$$E(S) = Kmne^{-\lambda S}$$

where m is the size of the query, n is the size of the target database, and S is the score of the alignment. K and λ are scaling constants [28]. The E-value gives the expectation of the number of matches with a score greater than or equal to S, given the sizes of the query and of the database. The lower the E-value, the more significant the alignment.

1.2.3 PSI-BLAST

PSI-BLAST [2] stands for Position Specific Iterative BLAST. As its name says, it is an iterative tool where the search results are refined through several consecutive BLAST searches. The first iteration of a PSI-BLAST search is executed in a similar fashion as a classical BLAST search of the query against the target database. From the results of this first iteration, a multiple alignment of the results with the query is constructed. Log-likelihood ratios are computed for all 20 aminoacids at each position of the alignment and tabulated in a Position Specific Scoring Matrix (PSSM). In the second iteration, the PSSM is used in place of the aminoacid substitution matrix to score alignments of the query with target sequences. The results of the second iteration are themselves turned into a PSSM which can further be used as input for a third iteration and so on. The search is iterated until the results of two iterations is identical, or until a maximum number of iterations is reached. Each iteration of PSI-BLAST takes slightly longer on average than a classical BLAST search. By taking into consideration more sequences than just the original query to find matches, PSI-BLAST is able to detect more distant homology than BLAST [2].

1.2.4 HMMER

HMMER [27] is not strictly speaking a sequence alignment search tool as it searches for alignments between Hidden Markov Models (HMM) and protein sequences (or *vice versa*). HMMs are a representation of multiple alignments akin to PSSMs. For each position, an HMM gives the probability of finding a given residue, of opening or inserting a gap, or of deleting the position. Figure 1.3 represents an example HMM model of an alignment. B and E represent respectively the beginning and the end of the alignment. I, M, and D states represent respectively Insertion, Match, and Deletion states. To each arrow in the model is associated a probability of going from one state to another. A random variable is associated to each I and M state giving the probability of inserting or matching a given residue at this position. HMMER has two main advantages compared to traditional score-based alignment methods. HMMs used by HMMER are a more precise model of a multiple alignment. For instance, they allow position specific gap opening and extending probability, instead of a general affine gap penalty. HMMs are also easier to interpret statistically. By optimizing the score of the alignment, score-based methods only take into consideration the maximum scoring alignment between two sequences. On closely related sequences, the similarity between the two sequences might be clear enough that only one alignment is possible. On more distantly related sequences however, it might not be clear how the two sequences should be aligned. A score based method, by only looking at the maximum scoring alignment, will miss this homology. HMMER, on the other hand, is able to recognize

that several alignment paths are possible and compute a global significance by marginalization over all possible alignments between the two sequences [13].

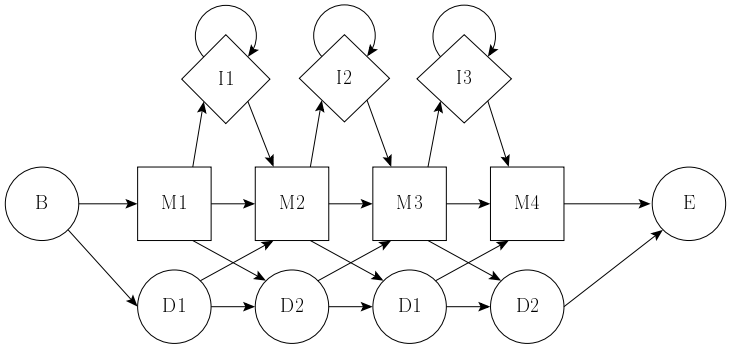

Figure 1.3: Example HMM representing a 4 position model.

Despite its theoretical advantages, HMMER has not replaced BLAST and PSI-BLAST up to now as the most used protein similarity search tool, due to its high computational cost. However the most recent version HMMER 3.0, released in March 2010, has seen great improvements both in terms of ease of use, adding for instance the possibility to carry out sequence-sequence alignments, and in terms of speed. This makes HMMER a viable alternative for large scale projects like ours.

1.3 Biological databases

1.3.1 UniProt

UniProt [52] stands for UNIversal PROTein resource. This it is a database of protein sequences. UniProt is split into two subsets, Swiss-Prot and TrEMBL. Swiss-Prot is the smaller of the two subsets. It contains manually curated and annotated sequences. TrEMBL contains sequences translated from gene sequences resulting from genome sequencing projects that are automatically annotated.

Figure 1.4 shows the evolution of the size of Swiss-Prot and TrEMBL over the years and illustrates well the discrepancy between the amount of information produced by genome sequencing projects and the quantity of data that can be manually reviewed: Swiss-Prot is almost 30 times smaller than TrEMBL. In the release 2011_06 released in June 2011, Swiss-Prot contained 529,056 sequences and TrEMBL 15,400,876 sequences.

1.3.2 Manually currated domain databases

SCOP

SCOP [38] is a manually curated hierarchical classification of proteins, inferred from three dimensional structures in the Protein Data Bank (PDB) [9]. SCOP groups domains in a 4 level hierarchical classification based mostly on their structural similarities. Domains are given identifiers of the form $A.B.C.D$ giving their classification at each level:

A defines the *class* of the domain. There are 11 different classes based mostly on the global secondary (two dimensional) structures of the domain.

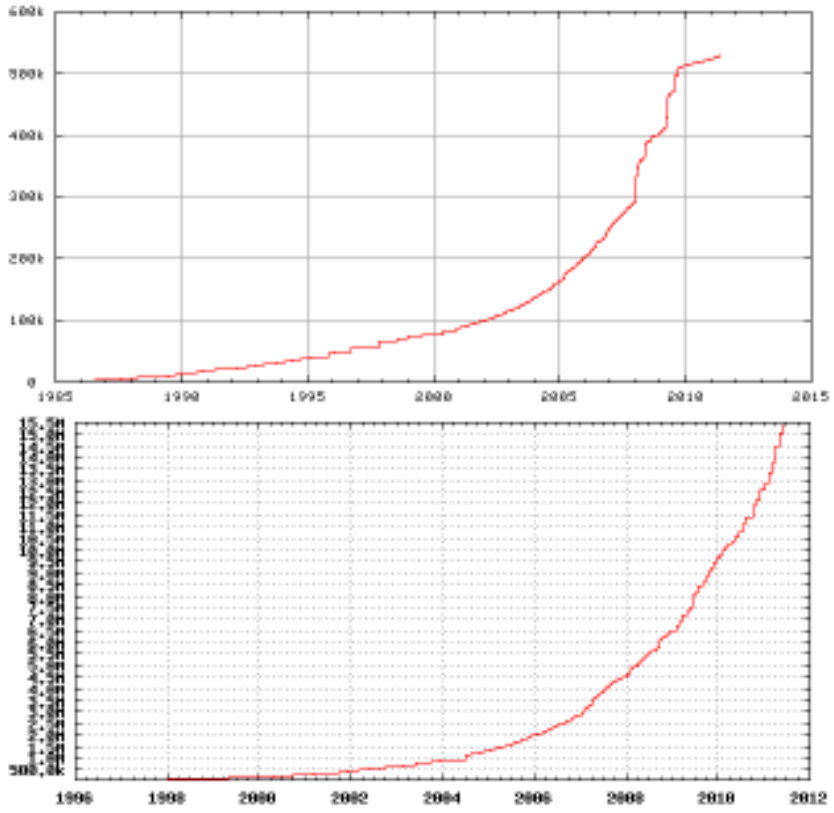

Figure 1.4: Evolution of the size of UniProt/Swiss-Prot (top) and UniProt/TrEMBL (bottom) over the years [70, 69].

B represents the *fold* of the domain. Domains in the same fold share major structural similarity. A common fold however does not indicate a common evolutionary origin.

C defines the *superfamily* of the domain. Domains in the same superfamily share enough structural similarities to suggest a probable homology.

D classify together domains into *family* that are strongly believed to be homologous. Most of the time this belief is backed by a high sequence similarity (more than 30% of identity) or strong structural concordance.

As of release 1.75, released in June 2010, SCOP classified 110,800 domains based on 38,221 three dimensional structures.

Pfam

Pfam [14] is a database of protein and protein domain families. Pfam is constructed in a semi manual process. Each family is represented by an HMM automatically constructed based on multiple alignments results that are manually curated. The set of manually curated families known as Pfam-A are complemented by automatically constructed families taken from the Adda database (see Section 1.3.3) called Pfam-B. The last release of Pfam, Pfam 25.0 was released in April 2011. it contains 12,273 families in Pfam-A and 142,303 families in Pfam-B. Pfam-A HMMs cover 53.86% of the amino acids in UniProt [66].

1.3.3 Automatically generated protein domain databases

Manual or semi-manual approaches are the best source of information from a qualitative point of view by the human expertise they convey, but their strength is also their biggest weakness. The best reported coverage is that of Pfam which covers about 53% of the amino-acids of known protein sequences, coverage which increases only slowly [14]. This gap might even widen in the future because of the exponential growth of sequence databases. For instance, the UniProt database [57] doubles in size approximately every two years. In addition, genome and metagenome data have shown that the universe of protein families is far from being completely represented in current data [58]. This gap in coverage has justified efforts to create methods that could infer domain boundaries and domain families automatically from homology relationships between sequences. MKDOM2, the method used to build ProDom falls into that category. In this section, we present two methods, ADDA and EVEREST, that have the same scope as MKDOM2 and that have been successfully applied on large datasets. We give a brief overview of MKDOM2 for comparison but a more detailed description of MKDOM2 is given in Chapter 2.

ADDA

ADDA [23] stands for the Automatic Domain Decomposition Algorithm. ADDA is a two step process which first defines the domains and then groups them into families. This process can be summarized into 5 steps as follows:

1. ADDA starts from an input dataset of protein sequences. For its most recent release, ADDA input dataset was composed of more than 1.5 million non-redundant sequences (*nrdb*).

2. The input set is compressed by grouping together sequences sharing more than 40% identity, yielding 249,264 representative sequences (*nrdb40*).

3. A global comparison of all the sequences of *nrdb40* is carried out using BLAST.

4. From the alignments found by BLAST, the ADDA algorithm determines domain boundaries in order to globally minimize the number of alignments that are split by a domain boundary as well as the number of residue pairs that are in the same domain without belonging to the same alignment. Figure 1.5 illustrates this mechanism on a sequence with two domains. In the latest release of ADDA, there were 2.7 million domains defined when reporting the domain boundaries onto the original 1.5 million original sequences.

5. Using the previously defined domains, a graph is created where vertices are domains and edges are BLAST matches. Each BLAST match, that is each edge of the graph, is verified via a profile-based alignment method between the two domains linked by the edge. Spurious edges are removed from the graph. Finally, each connected component remaining in the graph after checking all edges defines a family. The latest release of ADDA defined 123,000 families.

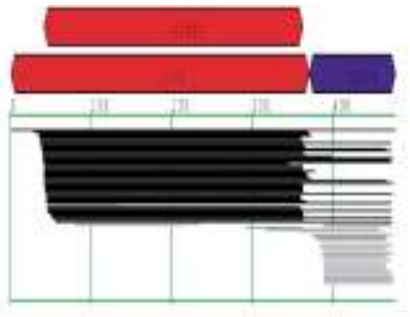

Figure 1.5: Domain decomposition of sequence EA6_ARATH from Arabidopsis thaliana in Pfam (top) and ADDA (middle), and BLAST neighbours (bottom) [23].

Everest

EVEREST stands for EVolutionary Ensembles of Recurring SegmenTs [42]. The EVEREST process is outlined in figure 1.6 and can be summarized in the following steps:

1. From an input dataset of sequences corresponding to all of SWISS-PROT (211,000 sequences in the latest release of EVEREST), a non-redundant database is defined by grouping together sequences that are aligned on more than 95% of their length with an E-value below 10^{-90}. The non-redundant database contained 125,000 sequences in the latest release of EVEREST.

2. Internal repeats are short repeated segments within a protein. Blast alignments with sequences containing repeats are hard to interpret. When one such case is detected, all but the first and last instances of the repeat are masked.

3. Based on the BLAST Results from Step 1, segments sharing similarities with other segments are detected and collected into a set of putative domains. At this stage the latest release of EVEREST defined 51,000,000 segments.

4. On each sequence, segments overlapping on at least 50% of their length are grouped.

5. Groups of segments are iteratively clustered together according to their similarity. The successive merges form a tree of potential clusters. The leaves of the tree are single groups while the root of the tree corresponds to all the groups merged into one cluster.

6. Using a statistical model [32] trained on intrinsic features of Pfam families (cluster size, similarity within cluster, variance of length of the domains in cluster, *etc*), all the potential clusters are tested.

7. For each cluster validated by the statistical model, an HMM is constructed.

8. A new database of segments is created by scanning the original database with the HMM created at the previous step. Steps 4 to 8 are repeated, starting from this new segment set.

9. HMMs which recruit the same segments are merged together to avoid overlapping families, leaving 20,230 final families in the latest EVEREST release.

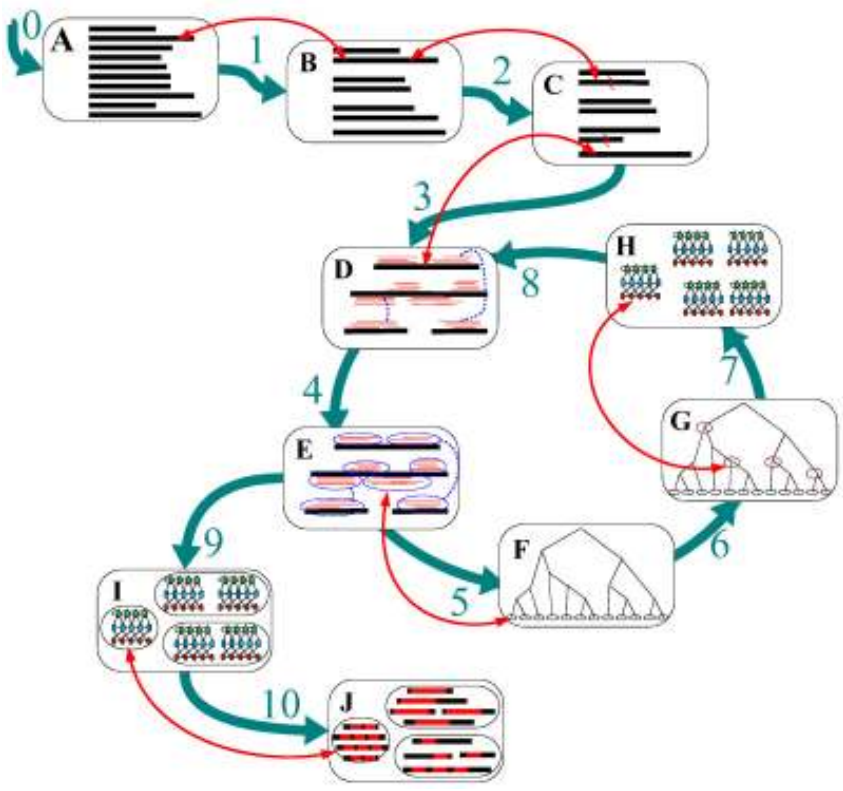

Figure 1.6: Illustration of the EVEREST process [42].

ProDom

ProDom [48] is a database of protein domain families built automatically using the MKDOM2 algorithm [19]. MKDOM2 is a greedy algorithm that is based on the simple heuristic that the shortest segment in a database is the most likely to be a mono-domain sequence. Until the database is empty, a family is created based on all the segments matching the shortest sequence via PSI-BLAST [2] and the family is removed from the database before the next iteration. The resulting domain family database, ProDom, has last been updated in 2008 and was based on a 2006 snapshot of UniProt. This last release was based on 2,001,128 sequences and defined 1,716,114 domain families.

The three methods presented here all have the same goal but employ different methods to reach it. All three methods base the construction of the domain families on homology relationships between the sequences. Both EVEREST and ADDA are based on an initial global all *vs* all BLAST to detect similarities between sequences. PSI-BLAST is a more sensitive algorithm than BLAST but when applied to multi-domain proteins, its iterative nature may generate chimeric results, and group together unrelated proteins. MKDOM2 on the other hand, by processing domain families one by one, is able to use PSI-BLAST instead of BLAST. ADDA makes up for the lesser sensitivity of BLAST by applying a careful single-linkage procedure to transitively expand the families. EVEREST on the other hand refines its initial set of families over several iterations, using a more sensitive profile based method.

EVEREST was last updated in 2006 and has never been tested on a database larger than Swiss-Prot. Both ADDA and ProDom have been able to cover all known protein sequences at one point. However ADDA was last updated using a 2007 snapshot of UniProt [14], and ProDom with a 2006 snapshot. Therefore, today no protein domain database covers the whole set of known protein sequences. This is due, at least for ProDom, to the prohibitive time that would be required to process all known sequences. Updating ProDom based on current data would take over 12 years using the current MKDOM2 algorithm. EVEREST and ADDA are composed of a series of steps that are, for the most part, composed of independent subtasks that can easily be carried out in parallel. Both EVEREST and ADDA already use parallel computing, at least in part during their construction. MKDOM2 on the other hand, computes families one by one, and is inherently a sequential process. For a new release of ProDom to be possible, a parallelization of its construction method is mandatory.

1.4 Parallel computing means

1.4.1 Shared memory platforms

Shared memory platforms are machines with more than one processor, or processor cores, sharing a common main memory. This type of platform is also often referred to as Symmetric Multi-Processor (SMP) or Unified Memory Access (UMA) systems because all the processors (CPU) have an equal access to the same memory (see Fig. 1.7). Most of today's computers, from laptops to servers are shared memory platforms. On an SMP system, a classic sequential program is only able to use one of the CPUs. In order to use several CPUs at once, a program must define independent subtasks or *threads* that can run in parallel on the different cores. Defining independent subtasks can be done either explicitly or implicitly through libraries such as OpenMP [7].

The parallelization of a program on an SMP has a relatively low overhead as all the threads have access to the same data through the main memory and there is no need to explicitly

transfer the data from one CPU to the other. The operating system ensures data consistency between the local cache of the CPUs and the main memory, it is however the responsibility of the programmer to ensure that two threads do not use the same data at the same time.

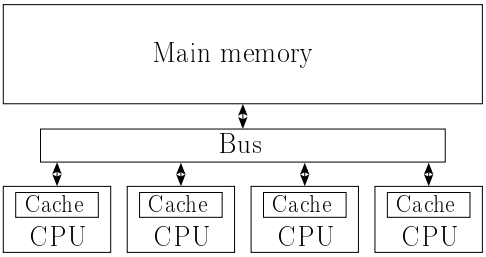

Figure 1.7: Schematic representation of an SMP system.

1.4.2 Distributed platforms

The most common form of distributed platforms are *clusters*. A cluster can be viewed as a set of individual machines linked together through a common network as illustrated by Figure 1.8. Each individual machine in a cluster is called a *node*. Clusters range from simple off-the-shelf machines linked together by a local Ethernet network and managed via software solutions such as Beowulf [49] to large supercomputers of several thousand machines linked together by fast proprietary interconnects such as the IBM Blue Gene supercomputers [50].

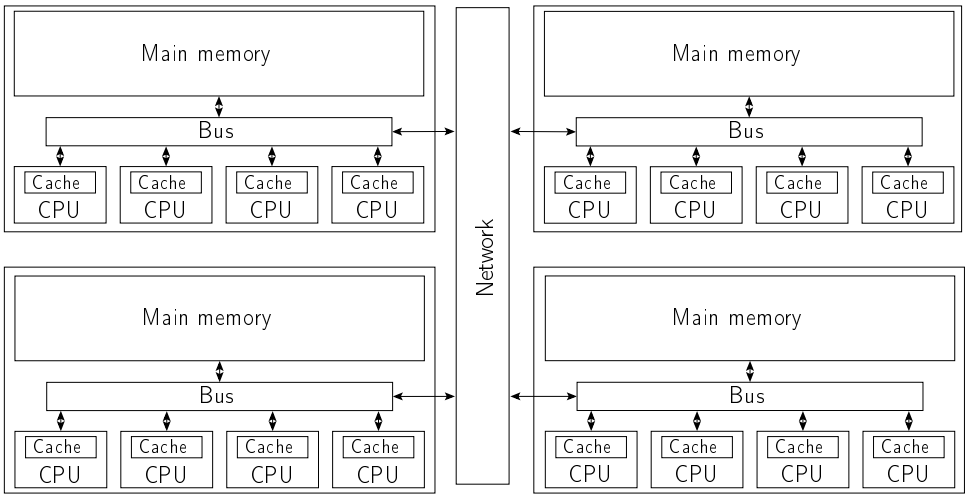

Figure 1.8: Schematic representation of a cluster.

On this type of platforms, the processors do not share a common main memory. Instead, data transfers between the different processors have to be explicitly defined by the programmer. Even fast interconnections between the nodes operate several orders of magnitude slower than a CPU. When designing an algorithm aimed at distributed platforms, one must take into account that communication costs may outweigh the benefits of increased parallelism. A distributed algorithm

must therefore use communications scarcely to be efficient and overlap communication time with computation time as much as possible.

Computing grids also fall in the category of distributed platforms. A computing grid is a set of clusters linked together by a wide area network. The main differences between clusters and grids from a practical point of view are: grids are more prone than clusters to be composed of heterogeneous resources both in terms of computing and networking resources; while nodes in a cluster commonly share a networked file system, clusters in a computing grid do not usually have a shared storage.

The two approaches, multi-threading and distributed computing, are not mutually exclusive. Distributed systems are usually composed of multiple SMP systems. It is therefore possible to use both levels of parallelism: a macro level, or *coarse grain* parallelism, to distribute large units of computations over a distributed platform, and a micro level, or *fine grain* parallelism, to further split each unit of computation on each node.

Besides the two classes of platforms presented here, there are other parallel platforms that allow finer parallelism such as General Purpose Graphical Processing Units (GPGPU) or coarser parallelism such as desktop grids. These platforms however are not addressed in this dissertation.

1.4.3 Platforms used throughout this work

Grid'5000

Grid'5000 [60] is a French research grid designed as an experimental testbed for large-scale distributed computing. The Grid'5000 is composed of 25 clusters representing almost 7,500 computing cores scattered over 10 sites interconnected through the RENATER education and research network [67]. Being an experimental platform, Grid'5000 is more customizable than a production platform usually is. However its heterogeneity both in terms of hardware and software makes it a less adapted platform for production and routine jobs.

CINES

CINES [59] stands for *Centre Informatique National de l'Enseignement Superieur*. CINES provides computing means for French universities and research. Computing hours are allotted twice a year to selected projects. We were granted 240,000 computing hours both in 2010 and 2011 on the *jade* cluster.

Jade [61] is a SGI ALTIX ICE cluster containing 23,040 computing cores, grouped in 2,880 nodes. Each node contains 2 Intel Xeon Quad-core processors (either E5472 or X5560) and either 30 GB or 34 GB of memory. Nodes are *diskless*, that is, they do not have local storage besides their memory. For persistent storage, jade relies on a 500 TB networked file system.

1.5 MPI

On distributed platforms, communications between the nodes have to be handled explicitly by the program. MPI [63] stands for Message Passing Interface and is an application programming interface (API) that facilitates the development of distributed applications and has become a *de facto* standard in high performance computing. A distributed application in MPI is a set of sequential or multithreaded processes identified by their *rank* and communicating by exchanging messages. The MPI standard defines various communication primitives to exchange messages

between processes of a distributed application. These communications can be grouped into three main categories:

Point to point blocking communications: a point to point communication is a communication involving two nodes, a sender and a receiver. A point to point message is defined by the rank of the sender, the rank of the receiver and a *tag* attached to the message. The sender initiates the communication by calling `MPI_Send` at some point in its execution, on the other side, the sender must call `MPI_Recv`. These calls are *blocking*, that is, both calls to `MPI_Send` and `MPI_Recv` return only once the message has been received. Blocking communications therefore have the side effect of synchronizing the two nodes.

Point to point non-blocking communications: non-blocking communications on the other hand can return before the message has been sent to allow processes to carry on their computation while the communication progresses, overlapping communication time with computation time. Non-blocking communications can be achieved through `MPI_Isend` on the sender side and `MPI_Irecv` on the receiver side. The `I` in the function names stands for immediate indicating that the calls should return as soon as possible to the caller before the communication has actually occurred. Non-blocking calls return a `request` to identify the communication late on. Using this request, processes must check that the communication has occurred through calls to `MPI_Wait` (blocking) or `MPI_Test` (non-blocking).

Variations of these functions allow a finer control on how these communications are handled for instance with `MPI_Ibsend`, the message is buffered on the sender side before the receiver is ready; `MPI_Issend` is unbuffered but the sender must not reuse or destroy the sent data before either `MPI_Wait` has returned or `MPI_Test` indicates completion of the transfer.

The efficiency of asynchronous communications is highly dependent on the implementation of the MPI library; for instance, if the library is not multi-threaded, the communications can only progress when the MPI library is accessed and regular calls to MPI_Test are in this case required to allow the communication to progress asynchronously. Both types of communications can be mixed for instance an `MPI_Isend` can be matched by an `MPI_Recv` on the other side, making the communication blocking only on the receiver side.

Collective communication: Collective communications involve a group of tasks, or *communicators* and are most of the time blocking communications. An example of such calls is `MPI_Bcast` through which one process can send a message to a set of nodes. Messages sent through `MPI_BCast` to n processes can be sent more efficiently than n individual messages by having the first receiving nodes re-send the message themselves.

Messages sent through MPI can be of any size. MPI usually deals differently with short and large messages (the limit between short and large being implementation defined). Short messages are sent directly and buffered on the receiver side. For larger message, a *rendez-vous* protocol is used, The sender first sends a short message to the receiver to ensure that the receiver is ready and has enough memory to receive the complete message before the actual message is sent. The MPI library may enforce a flow control mechanism and temporarily limit or block new messages to ensure that a node is not overwhelmed by incoming messages.

The MPI standard defines Fortran, C and C++ bindings. However C++ bindings are marked for deprecation in the next version of the standard [64]. There are several open source

implementations of the MPI standard such as MPICH2 [21] and OpenMPI [16]. Additionally, hardware manufacturers usually provide MPI implementations tailored for their machines, sometimes based on open implementations.

Chapter 2

Parallelizing the construction of ProDom

In the previous chapter we introduced the problem of protein domain family clustering and the methods available to solve it including MKDOM2, the algorithm in charge of updating the ProDom database. The interest for ProDom as a data source for evolutionary biology is still strong. Indeed, each year, the ProDom web server handles 60,000 unique visitors, and more than 1,500,000 queries. The survival of ProDom as a viable reference is challenged by the massive amount of new sequences pouring in every day. The most recent release of ProDom is called ProDom 2006.1 because it was built against the set of all complete protein sequences available in UniProt in 2006. It took 15 months for MKDOM2 to create the domain family clustering. In the meantime, UniProt kept on growing and upon its release in late 2008, ProDom 2006.1 was already covering less than half of the available sequences. MKDOM2 is now unable to maintain ProDom up-to-date. Given the increase in input data, computing a new ProDom release today would take more than 10 years of computation, an unrealistic duration for a sequential algorithm.

To be able to continue building ProDom on up-to-date and exhaustive data, we have designed MPI_MKDOM2, a new parallel algorithm. Our aim in designing MPI_MKDOM2 was twofold: 1) from the biological point of view we did not want to compromise on the quality of the output, and thus decided to stay as close as possible to the behavior of MKDOM2; 2) from the computer science point of view we wanted an efficient distributed algorithm capable of harnessing the processing power of distributed computing platforms to compute ProDom in a reasonable time frame. Satisfying these two objectives appeared to be contradictory because of the assumptions the MKDOM2 algorithm relied upon. MKDOM2 relies on the different parts of the computation occurring in a certain sequence and, therefore, some computations cannot be carried out in parallel. Moreover, the running times of the different parts of the computation are unpredictable and highly variable. It is therefore extremely challenging to design an efficient scheduling of these computations.

Section 2.1 is dedicated to the static and dynamic analysis of MKDOM2. In Section 2.2 we explore two possible alternative parallelization strategies and detail their pros and cons. In Section 2.3 and 2.4 the most promising parallelization approach is refined to address both biological and computational difficulties. Finally the last two sections are dedicated to experimental results, in Section 2.5 we test MPI_MKDOM2 on small datasets to characterize its parallel performance and in Section 2.6 we further show the validity of our approach by using MPI_MKDOM2 to create a new release of ProDom from all available sequences in UniProt as of July 2010. The quality of the results is assessed in Chapter 3.

2.1 The MKDOM2 algorithm

2.1.1 Algorithm overview

MKDOM2 is a greedy algorithm, that defines domain families iteratively one after the other, removing domain families from the database as it goes, until the database is empty [19]. Each iteration can be further detailed into 5 steps as follows:

1. Protein domains considered in ProDom have a minimum length of 20 residues: MKDOM2 discards all shorter sequences.

2. The shortest sequence is selected under the assumption that it contains only one domain. The more comprehensive the protein database, the stronger the assumption.

3. Proteins often contain repeated segments [34], in particular tandem repeats (back to back occurrences of homologous segments). MKDOM2 tries to detect these cases and if a repeat is found, the repeat motif is selected in place of the original sequence.

4. The selected sequence (or repeat unit) is used as a query to search the database. Segments of protein found homologous to the query are defined as domains. A new domain family is defined that contains the query and the homologous segments.

5. All the domains from the newly defined family are removed from the database. Domains are segments of sequences. Removing a domain from one of the extremities of a sequence will shorten this sequence. Removing a domain from the middle of a sequence will split this sequence in two new sequences, on the left and right ends of the domain. Both of these operations create shorter sequences that may become the new shortest sequence of the database.

The algorithm is summarized in pseudo code in Algorithm 1. Figure 2.1 illustrates the behaviour of the algorithm over one iteration when the query finds no results, some results, or if the query is a repeat.

2.1.2 Runtime analysis of MKDOM2

The goal of the runtime analysis – or profiling – of a program is to find how much time is spent in the different parts of this program. This is generally achieved by recording the time at which a program enters and leaves any function. While considerably slowing down the actual processing, the time-keeping mechanism should not affect the relative weight of the different components of a program. Knowing how time is spent across a program helps identifying *hotspots*, that is, parts of the program that are particularly time consuming and on which optimization and parallelization efforts should be focused. MKDOM2 being implemented as a Perl script, the profiling was carried out using Perl's `Devel::DProf` profiler. Figure 2.2 shows the abridged results of the profiling of MKDOM2 on five different database sizes.

We profiled MKDOM2 running against 4 subsets of the SWISS-PROT database of increasing sizes. The profiling results reveal that two functions `PsiBlast` and `UpdateDB` represent most of the computational cost of the program. With the largest database tested, `PsiBlast` and `UpdateDB` account for almost 90% of the total computing time of MKDOM2. The other functions, for instance for parsing the PSI-BLAST output, verifying that the each family does not contain duplicated segments or writing out the output, tend to account for proportionally less of

Algorithm 1: MKDOM2

Input : sequences, a collection of protein sequences
Output: families, a collection of protein domain families

Data: query, a protein sequence
Data: family, a set of protein segments

while sequences *not empty* **do**
1. **foreach** *sequence* seq *in* sequences **do**
   - **if** seq *shorter than 20 residues* **then**
     - remove seq from sequences
2. query ← shortest sequence in sequences
3. **if** query *contains a repeat* **then**
   - query ← repeat motif
4. family ← psiblast(query, sequences)
   add family to families
5. **foreach** *segment* seg *in* family **do**
   - remove seg from sequences

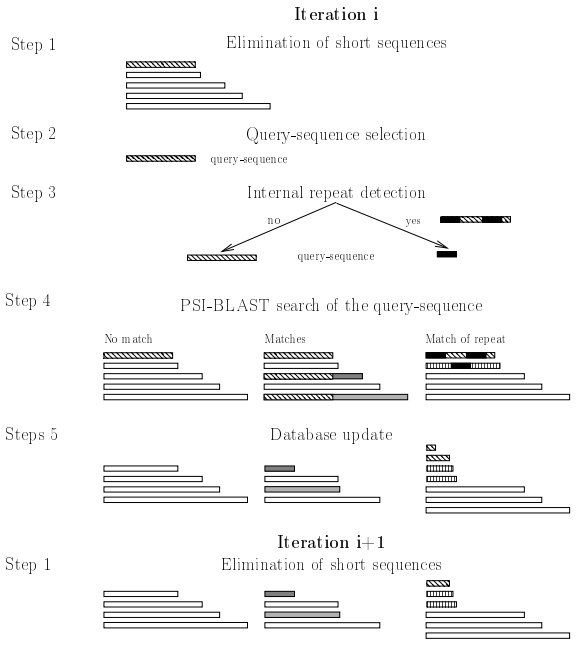

Figure 2.1: Illustration of an iteration of MKDOM2 [19].

the total computing time as the database grows. `PsiBlast` carries out Step 4 of each iteration of MKDOM2. As its name indicates, it uses PSI-BLAST [2] to search the database for segments that are homologous to the query. `UpdateDB` corresponds to Step 5 of the iterations. Its objectives are to remove the newly created families from the database, to sort the sequences according to their size and to reformat the new version of the database for PSI-BLAST. Any attempt to significantly affect the running time of MKDOM2 should investigate the optimization and parallelization of these parts of the algorithm.

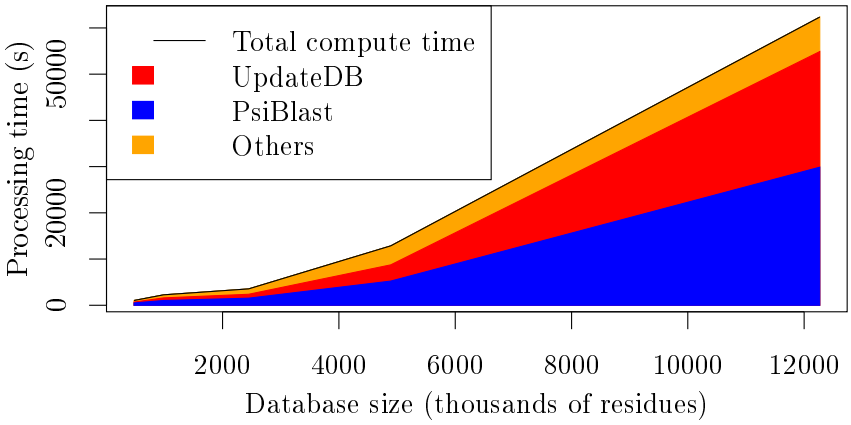

Figure 2.2: Evolution of the processing time of MKDOM2 as a function of the database size.

2.2 Parallelization strategies

The profiling of MKDOM2 revealed that the most time consuming part of the algorithm is the PSI-BLAST search. We see two possible parallelization approaches:

Internal parallelization: We call internal parallelization the parallelization of PSI-BLAST itself, that is, for each iteration, allow PSI-BLAST to use 2 or more cores to process a query.

External parallelization: The external parallelization corresponds to a scenario where we allow several queries to be processed in parallel, each PSI-BLAST instance using one core.

These two approaches are not necessarily mutually exclusive. Depending on there respective merits and drawbacks, we may decide to use one, the other, or both combined.

2.2.1 Parallelizing PSI-BLAST

Parallelizing PSI-BLAST means allocating 2 or more cores for PSI-BLAST to process each query. Such an approach would have the great advantage, from the biological point of view, to leave untouched the overall structure of the algorithm and thus the nature of the heuristics. No change would be made in the order of the queries, they would only be processed faster. The parallel and sequential versions of the algorithm would deliver the exact same output.

This strategy can quickly be tested since PSI-BLAST already has an option that lets the user set the number of threads. We tested this option using query sets and the target database from the RefProtDom [18] evaluation dataset to test the acceleration possible when using several threads. Figure 2.3 shows the observed speedup, that is the acceleration factor with respect to the sequential execution, and the efficiency, the speedup divided by the number of resources used, as a function of the number of threads. PSI-BLAST multi-threading yields suboptimal results, reaching only a 1.60 speedup with 8 threads (with 8 cores available). One may say that these results are due to a sub-par implementation and do not reflect the potential speedup of the strategy itself. Looking more closely at how PSI-BLAST functions below, it appears that it would be really difficult to reach a significant speedup with this strategy.

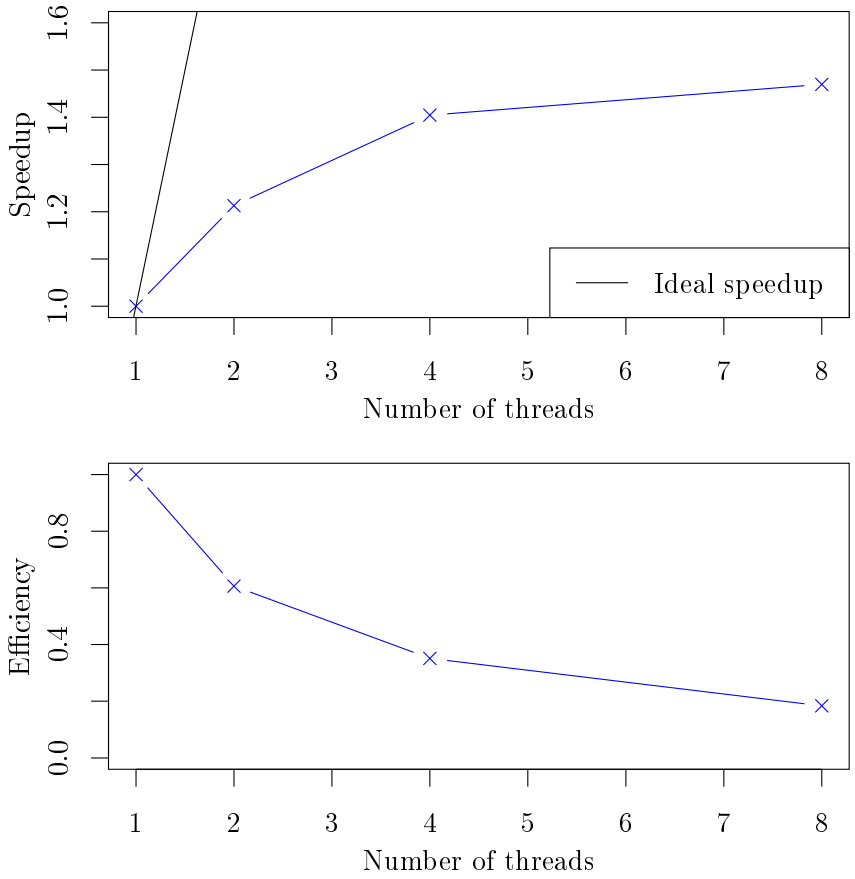

Figure 2.3: Speedup (top graph) and efficiency (bottom graph) of PSI-BLAST as a function of the number of threads used for the processing of the hard and sampled sets of queries from the RefProtDom dataset [18]. The machine used for the tests had 8 cores and PSI-BLAST was setup up to stop after 2 iterations.

A typical execution of MKDOM2 is composed of a large number of iterations that are in average quite short. For instance, MKDOM2 needed 88,788 iterations to process a database containing 69,621 protein sequences, each iteration lasting on average 0.58 seconds. A parallel version of PSI-BLAST would therefore be a very fine-grain parallelization. For such a parallelization to deliver significant speed-ups, the cost of the parallelization must be really low.

If the load is not almost perfectly balanced, the loss in efficiency induced by the parallelization will quickly outweigh its benefits. Unfortunately, the structure of PSI-BLAST makes any parallelization attempt prone to poor load balancing. Indeed as detailed in Section 1.2.3, a PSI-BLAST search is also an iterative process: it is a succession of BLAST searches. The first iteration is a classic BLAST search of the query sequence against the database. After this first step, the results of the search are merged into a position specific scoring matrix (PSSM) based on the probability of aligning a given residue at each position of the query. For the second iteration, the PSSM is used as query instead of the sequence. The results from this second iteration are again turned into a PSSM. The process goes on until either two successive iterations generate the same result or until a maximum number of iterations is reached (in MKDOM2, the maximum number of PSI-BLAST iterations is set to 10). The construction of the PSSM is a strong synchronization point between all the threads: the next iteration cannot start without the new PSSM and since this PSSM summarizes the results of the current iteration, all the results must be available in order to create it. While the longest thread finishes computing the current iteration on one core, all the other cores are left idle because nothing can start unless all the results are available. Keeping the time wasted at the end of each iteration to a minimum would require a good load balancing between all the threads so that they all terminate at approximately the same time. Unfortunately, the duration of each thread depends on how many alignments it will have to perform which cannot be predicted in advance [17]. No matter how well implemented this solution is, such a tightly coupled parallelization with no guarantee on load balancing is unlikely to deliver useful speed-ups. With a shared memory parallelization yielding such poor results, chances are that a distributed approach would perform even worse because of communications between different machines. It can be noted that the distributed implementation of the BLAST tool suite, mpiBLAST [8], does not include PSI-BLAST for similar reasons [65].

2.2.2 Running several iterations in parallel

In the second strategy, the parallelization is done outside of PSI-BLAST. Instead of distributing the load of one query over several cores, we allow several queries to be processed at the same time. This parallelization theoretically gives us more room to maneuver since the atomic operations we would be running in parallel are larger and less coupled. If the successive iterations were totally decoupled, MKDOM2 would be an embarrassingly parallel problem: there would be no restriction on how many iterations can be run in parallel, and the only decision to be made would be how to schedule these iterations to best absorb their variations in completion time and to minimize the completion time of the whole process.

MKDOM2 unfortunately relies on the creation of the families one after the other in a greedy way: departing from the sequential logic of the algorithm may have repercussions on the results. The sequentiality of the iterations in MKDOM2 has two advantages: firstly it ensures that each family is created from the most likely mono-domain sequence, and secondly, by updating the database each time a new family has been defined, it guarantees that families cannot overlap. If several iterations are to be run in parallel, our scheduling strategy must not only address load balancing problems, it must also take into account that, from a biological point of view, some scheduling strategies may be better than others. Yet with all the concerns raised above, this strategy is the only one worth trying as the internal parallelization has proven unfit for the large scale distributed processing.In the next section we discuss these problems in detail and outline ways to handle them.

2.3 From a strategy to an algorithm

Of the two possible parallelization strategies only one, which consists in processing several queries at the same time, offers the prospect of significantly reducing the wall-clock time of MKDOM2. In its simplest form, this solution can be viewed as a data parallel version of MKDOM2 following a master-worker scheme: One process, designated as the master, is in charge of allocating queries to the worker processes that run PSI-BLAST searches and send results back to the master process. Algorithm 2 illustrates the behaviour of the master node in a naive version of this scheme. Unfortunately, as mentioned above, MKDOM2 cannot be reduced to an embarrassingly parallel problem. Parallelization should be implemented with attention to several considerations that are important for computational performance or result quality:

Order of the queries: By always selecting the shortest sequence at each iteration, MKDOM2 was ensuring that families were most likely created from a sequence that contains only one domain. By selecting several queries at once, we may alter this property in a significant manner.

Load balancing: This problem may arise even in an embarrassingly parallel algorithm but in MKDOM2 it is especially stringent. Running times of different PSI-BLAST instances may vary by several orders of magnitude.

Overlapping families: By creating several families in parallel, we will encounter situations that the sequentiality of MKDOM2 prohibited: Two families created in parallel may claim the same part of the same sequence, at which point we must decide which of the two families has the priority over the other.

The above problems posed by this parallelization strategy have been addressed and have shaped the way we designed our parallel algorithm. The database updates which represent the second most time consuming part of the algorithm have not yet been addressed at this stage of our analysis. They must also be taken into account for the algorithm to be efficient.

Algorithm 2: Master in a naive master-worker parallelization of MKDOM2 over n processes.

while *Database is not empty* **do**
- Select $n-1$ sequences
- Send 1 sequence to each of the $n-1$ workers
- Gather the results
- Resolve the potential conflicts between the overlapping results
- Update the database

2.3.1 Order of the queries

While running several queries in parallel, we would still want our parallel algorithm to stay as close as possible to the sequential one: sequences running in parallel must be chosen among the shortest sequences of the database. Indeed, several of these sequences are likely to be mono-domain sequences, especially on large databases. MKDOM2 already makes an arbitrary decision on which is the best candidate to be a single domain in case of a tie. If for instance at a

given iteration, the shortest segment is 50 residue long, MKDOM2 still has to choose which of the 50 residue-long sequences will be used to create the next family. Therefore, even by selecting many sequences at once, this parallel strategy would not stray far from the original heuristics.

Under the original MKDOM2 algorithm, the processing of a query $\mathcal{S}_1$ can lead to the apparition of a new shortest sequence $\mathcal{S}_2$. Following the logics of MKDOM2, $\mathcal{S}_2$ is now the most likely sequence to be mono-domain, it should therefore be the next query. In a parallel execution, the query $\mathcal{S}_1$ is processed among several other queries that may be longer than $\mathcal{S}_2$ and that, in effect, are overtaking $\mathcal{S}_2$ with respect to the sequential logics. In order to stay true to the sequential heuristics, the parallel algorithm should cancel the other currently running iterations and start over from $\mathcal{S}_2$. This approach was tested in a first parallelization attempt carried out in 2004 by Blanquart [5] but unfortunately this parallelization strategy could only provide a maximum speedup of 1.71. It is therefore not possible to go significantly faster than MKDOM2 while keeping exactly the same constraints. This *a priori* forbids that a short sequence, produced by the processing of the first n query-sequences, be processed before these n query-sequences are all processed and their results removed from the database. Therefore, a parallel execution will not process query-sequences in the same order as the original MKDOM2 algorithm. This will obviously have an impact on the solution that should be minimized.

2.3.2 Load balancing

We discarded the internal parallelization of PSI-BLAST on the basis that the tight coupling between the threads and the unpredictability of the running time of each thread made it impossible to correctly balance the load within each of the successive BLAST searches across different workers. Parallelizing outside of PSI-BLAST, we now need to balance the load of several instances of PSI-BLAST. The running time of PSI-BLAST is at least as unpredictable as the running time of its internal iterations. In fact, there may be several orders of magnitude between the shortest and the longest running time. Figure 2.4 shows the duration of PSI-BLAST instances while processing a database of 556,964 sequences: the average duration of a query was 3.21 s, the median 2.59 s and the maximum 266 s. However, contrary to the first strategy, our iterations are in this case independent of each other. Provided we can handle conflicts between results afterwards, we are not required to wait for the completion of all the currently running PSI-BLAST searches to start a new one. Our algorithm will therefore absorb the irregularities in PSI-BLAST run times by desynchronizing the workers, the master providing them with a new query as soon as they finish one.

In this asynchronous scheme, the first workers to finish do not have to wait for the more loaded workers to complete their queries. However they still have to wait for the master to select new queries before resuming work. In order to limit the time workers need to wait for the master, we would like the master to start selecting the next queries before the workers have finished computing their allotted task. In addition, a worker processing a short query that does not find any match will return its result almost immediately, not giving the master enough time to select new queries. In order to avoid this behaviour, the master needs to send batches of several queries to each worker. When one of the worker is already halfway through its allotted batch, the master starts selecting new queries for all the workers. The query selection by the master now happens before it receives all the results from the workers: it anticipates that all workers will finish in a near future and will soon need new queries. It might be possible however that one of the workers will spend a large amount of time processing one of its queries. To avoid queuing queries on an already overloaded worker, the master will not send new queries to a node

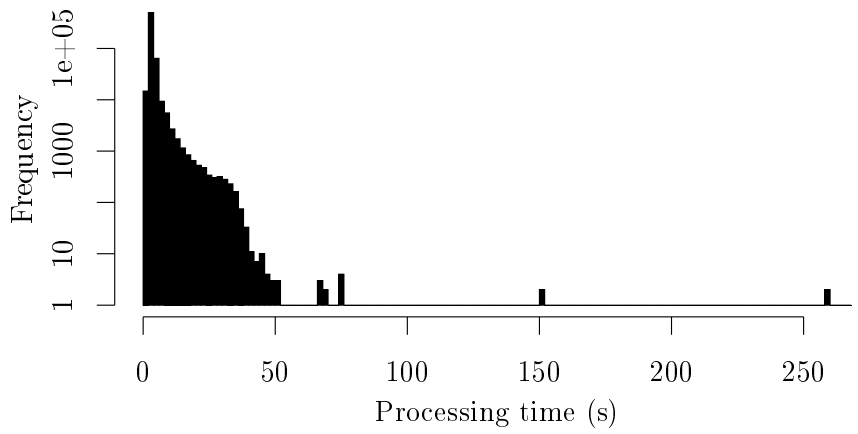

Figure 2.4: Distribution of the processing times of queries for the processing of a 556,964 sequence database.

which still has a full batch of queries left to process.

Figure 2.5 illustrates the differences between a synchronous (top of the figure) and an asynchronous (bottom of the figure) execution. The waiting time at the beginning of the first iteration is unavoidable in both schemes, but from the second iteration onwards, the inactivity period for the workers and the total wall-clock time is reduced by the asynchronism between all the workers and between the workers and the master.

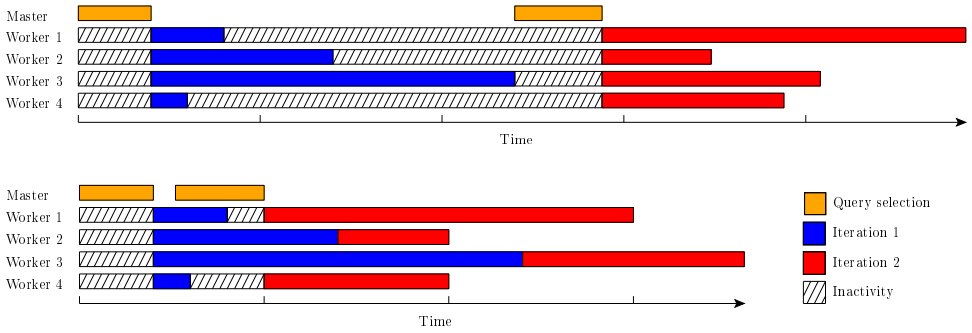

Figure 2.5: Illustration of the difference between synchronous (top) and asynchronous (bottom) execution.

The target database must be regularly updated to remove the segments that have been identified as domains. If the database is centralized and the work of the workers desynchronized, at any time there may be a worker processing a query and using the database. In order to update the database, the master would have at some point to stop all the workers which would reintroduce a synchronization point between the workers. Asynchronous workers are therefore incompatible with a centralized database. In order to allow the workers to work on up-to-date databases without stopping all the workers, we need to give each worker its own copy of the database. The worker will receive, along with a new query, a list of all the families that have been defined since its last update. The worker will need to remove the families from its own database prior to starting computing the new query. Duplicating the database, and therefore

the database updates, on all the nodes may seem like an unnecessary duplication of computation and a waste of storage space. However, with a single shared database, all the workers would have to stop before the update could start and they would have to wait while the master performs the update. The workers would be unable to process queries during the update but also potentially prior to the update as well, waiting for other workers to finish. With one database per worker, each worker still cannot run queries while it is updating its database, but it does not have to wait for all the workers to stop before performing its update. Therefore, even if replicating the database requires more work from the workers, in term of wall clock time however, it cannot be slower than a centralized database.

While we try to keep the workers as busy as possible processing queries, we want exactly the opposite for the master. The master node plays a pivotal role in a master worker scheme. If the master is overloaded, it can quickly become a contention point for the whole process. The less work the master has to do, the more workers it can handle. By delegating the database updates to the workers we also unload the master node from a costly task. In fact, by giving each worker a copy of the database, we remove the last reason for the master to actually have a copy of the database. To keep track of the remaining sequence segments in the database and define updates, the master node only needs metadata defining these segments: a sequence identifier and start/end positions (or start position and length) suffice to uniquely identify a segment.

An asynchronous computation scheme also requires asynchronous communications between the nodes. Using synchronous communications in an asynchronous execution scheme is prone to trigger deadlocks. When two nodes want to communicate with synchronous communications, both sides must be ready before the communication can occur. For instance, if the master node has to send new queries or an update message to a worker w_i, it has to wait for the worker to be ready to receive. If in the meantime w_i tries to send a query result back to the master, both nodes get stuck indefinitely waiting for the other node to finish its current operation. With synchronous communications, exchanges between the node must be sequentially scheduled in order to avoid these situations. Even in a deadlock free scheme, using synchronous communications still has some unwanted consequences. If w_i is currently processing a query when the master node starts sending it a message, the master node will effectively be blocked until the end of the processing on w_i. If other workers in the meantime need to send their results back to the master, they in turn get blocked as well. Synchronous communications, by requiring a fixed sequential scheduling of the transactions induce synchronization of the computation between the nodes.

2.3.3 Dealing with overlapping families

By computing several families in parallel, we are likely to run into a situation where two families computed in parallel claim a common part of the database. This situation may occur for two different reasons:

- The conflicting queries may both be representatives of the same domain family. In that case, both queries will gather identical or highly similar result sets and each query will appear in the result set of the other. This situation may occur fairly regularly. Indeed, each time we select new queries, we choose sequences among the shortest that have exactly or almost the same length. Variations of a homologous domain are likely to be similar in length and to become eligible as queries at around the same time.

- Overlaps between result sets may also be fortuitous; two queries belonging to two different domain families may still overlap over a few residues if a member from the first family

is neighbouring a member from the second family in the same protein sequence. Indeed domain boundaries are not sharply defined. In MKDOM2, an inferred domain ends where the alignment ends. However this alignment might extend across the true domain boundary due to incidental similarity, creating an overlap with the neighbouring domain. An illustration of this scenario is given in Figure 2.6. Because of its sequential nature, the MKDOM2 algorithm was implicitly taking care of these peculiarities (yet in an arbitrary way). In a parallel execution we will have to deal with this problem explicitly.

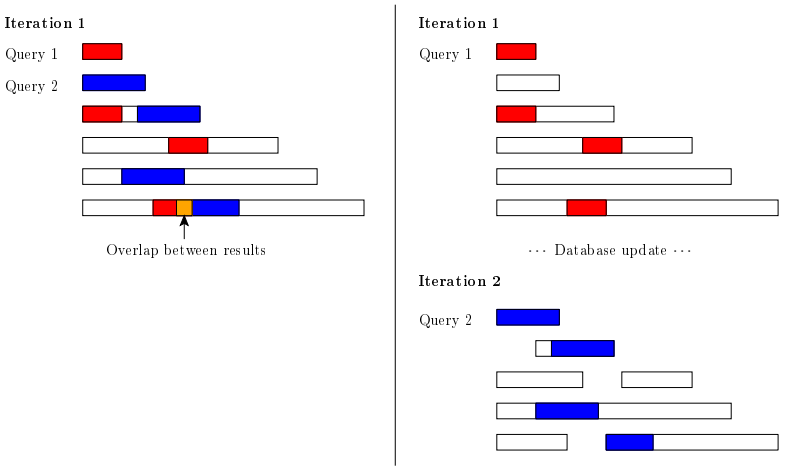

Figure 2.6: Illustration of a conflict between two results. In the sequential version (right-hand side), the database update prevents the potential overlap that leads to a conflict in the parallel version (left-hand size).

Despite these situations, we still expect our end results to be a partition of the protein space into non overlapping families. Even in a parallel processing, queries are still selected in a sequential fashion by the master, queries are therefore ordered and we use this ordering to solve the conflicts. Results are prioritized according to the age of the queries. Indeed, we outlined previously that queries may have widely varying processing times, which implies that the master may receive the result of query $\mathcal{S}_2$ before that of query $\mathcal{S}_1$ even if $\mathcal{S}_2$ was selected after $\mathcal{S}_1$. In the case of a conflict, we want to give the priority over the conflicting residues to the result of the oldest query, that is, to the query that was selected earlier by the master. The master must therefore maintain a waiting list of results to queue the results and process them in an orderly fashion, no result can be taken into consideration unless all the results of the previous queries have been received and processed.

In the case of an overlap due to homology, one of the results is lost for good since both results reflect the same underlying homology relation between the segments. In a sequential processing only the fist selected sequence would be processed so it seems logical to discard the result of the second one. If the case of an overlap due to neighbouring domains, one may think that in order to avoid recomputing the second family, shaving the few compromising residues off the most recent query would be a fair resolution of the conflict. In practice however, this solution is not applicable. If the overlap is indeed accidental, a few residues removed from a segment may make the quality of its match with the query fall below the threshold above which we consider

the match to be relevant. Using PSI-BLAST makes the matter even worse: if this segment was considered in the construction of the PSSM, it may have influenced other matches that would not have been found were those residues absent from the database. In conclusion, even with an incidental match we have to consider the whole result as tainted and discard it completely. The same query may be processed again once the database has been updated and the conflicting residues have been removed.

Each time we encounter a conflict, regardless of the cause, we must throw away one of the results and the compute time spent creating that results is therefore wasted. If it is an incidental overlap, the situation is even worse since for only a few overlapping residues, we must recompute the discarded result again. All this time spent computing families that are not part of the end results is harmful to the performance and these situations should therefore be avoided if we want our parallel algorithm to be efficient. Avoiding all these overlaps however, would require knowing the results in advance. Yet with some additional information on the sequences, we may be able to forecast potential conflicts sufficiently well to prevent most of them.

Given two sequences, we would like to know if they are likely to match the same parts of the database through PSI-BLAST. For this we rely on a global comparison of all sequences in the initial database using BLAST. The results of this BLAST search are summarized by a logical matrix $\mathcal{A}$, called the adjacency matrix, where $\mathcal{A}_{\mathcal{S}_i,\mathcal{S}_j} \neq 0$ if sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ were found to match locally. Using this matrix as a guide, we only select queries to run in parallel if they do not share similarity in this first BLAST search: if $\mathcal{A}_{\mathcal{S}_i,\mathcal{S}_j} \neq 0$, $\mathcal{S}_i$, $\mathcal{S}_j$, or any subsequence of $\mathcal{S}_i$ or $\mathcal{S}_j$, may never be selected as queries at the same time. Since a PSI-BLAST search generates broader results than a classical BLAST search, the all-against-all search cannot absolutely guarantee that the results of non-homologous queries will not overlap. This additional information, however, ought to be sufficient to ensure that the occurrence of these overlaps is infrequent enough not to have a significant impact on the performance of the parallel execution. We outlined in 2.3.2 that queries must be processed asynchronously to balance the load across the workers. The queries selected at a given time must therefore not only be independent from each other, they must also be independent of all the previously selected queries which are either still running or whose results have not yet been validated and removed from the database.

In our scheme, two sequences $\mathcal{S}_1$ and $\mathcal{S}_2$, which are fragments of homologous sequences, are considered homologous even if $\mathcal{S}_1$ and $\mathcal{S}_2$ are not themselves homologous. This has two unpleasant consequences towards the end of the processing when most of the sequences remaining in the database are fragments of larger sequences: 1) this may artificially decrease the number of potential query-sequences thus leading to partial starvation (not enough runnable query-sequences to harness all the platform resources); 2) by forbidding the selection of some small queries for reasons that are not applicable any more, the adjacency matrix might force us to select bigger queries than we would actually need. Recording the positions of the matches on the sequences could also help reduce the number of false positives. However the information on matches is kept as basic as possible for performance. Indeed query selection is performed by the master that should be as lightly loaded as possible, as outlined previously in 2.3.2. The adjacency matrix in its binary form is already a massive amount of information, scaling quadratically with respect to the size of the database; having the master handle this most basic information efficiently enough to avoid hurting the global efficiency of the algorithm is already an implementation challenge, adding more information may do more harm than good if it slows the master down significantly.

Running all the sequences through BLAST may be seen as a very expensive preprocessing phase, almost as expensive as the whole computation we want to parallelize. However contrary to

the MKDOM2 algorithm, this all-against-all BLAST search is embarrassingly parallel and can therefore be trivially parallelized. Furthermore, this search uses a simple BLAST search which is less time-consuming than a PSI-BLAST search. Therefore, even if this preprocessing has a large computational cost, it enables us to reach our goal: to very significantly decrease the wall-clock time needed to produce the ProDom database (including the time needed to perform this preprocessing). Moreover the global comparison of all sequences is a common computation in bioinformatics that is required for other applications: its cost can be shared with other projects. For instance, in the prospect of routinely creating new releases of ProDom, we may envision to reuse the results of the global comparison required to build protein families in the HOGENOM database [41].

Optimizing database updates

The database updates in MKDOM2 are there to ensure that once a protein segment has been defined as a domain in a family, it can never appear again in another search result and therefore be part of another family. In MKDOM2 this is done by recreating a new version of the database where the previously defined domains have been removed and by reformatting the new version of the database into BLAST format. It would seem more efficient to amend the existing BLAST database than recreating a new version at every iteration, but writing into an already formatted BLAST database is not officially supported. For this matter, MKDOM2 use of PSI-BLAST is at odds with the way BLAST tools were intended to be used. Reformatting a database in BLAST format is a costly process that reorganizes the data for PSI-BLAST efficiency. This is deemed worthy because the target database is assumed to be long-lived, so that many queries may be performed on a database once it has been formatted. The cost of database formatting is therefore balanced by the time saved every time PSI-BLAST uses the database. However in MKDOM2 the lifespan of a database is typically only one query, which makes reformatting the database a burden.

Although updating the database is not officially supported, it is not impossible. Formatting a set of sequences into BLAST format creates three new files: a data file containing the sequence data, a header file containing metadata (*e.g.*, sequence identifiers) and an index file containing the start positions of each sequence and its metadata in the data and metadata files, respectively. Modifications can be applied to the data file provided these modifications do not require changing the length of the sequences. Taking advantage of this fact, we designed a masking mechanism that relies on the scoring mechanism of PSI-BLAST to ensure that masked segments can never appear again in the result of a query against the database. With the default parameters, which we use in MKDOM2, an alignment stops if its score drops by 25 points or more without going up again. In the amino acid alphabet, one symbol, the end of translation or '*', scores particularly badly when matched with any other residue than itself. As '*' denotes the end of the translation, it does not correspond to an actual amino acid but is a counterpart for STOP codons in DNA. In the standard BLOSUM62 [24] scoring scheme used by PSI-BLAST, a '*' is worth -4 points against any residue other than itself. Since domains defined by MKDOM2 are composed of 20 or more consecutive residues, replacing the residues of a domain by a series of '*' gives it an alignment value of at least -80, which is more than enough to ensure that the domain cannot appear again in a search result. Over the processing of a database by MPI_MKDOM2, using the update mechanism requires as many write operations as one formating of the full database. This mechanism implies that MPI_MKDOM2 must hold an index to know where to apply the modifications in the data file. This functionality however is provided "for free" as an index is

already created by the BLAST formatting and MPI_MKDOM2 can reuse the same index.

2.4 MPI_MKDOM2 outlined

In the previous section we presented the most important points of our distributed algorithm, MPI_MKDOM2. In this section we summarize our algorithm and present MPI_MKDOM2 in its complete form on the master and worker sides.

2.4.1 Notations

We note $\mathcal{S}$ the initial list of protein sequences $\mathcal{S}_1, \cdots, \mathcal{S}_n$. We use $\mathcal{S}_i^{b,e}$ to identify a segment on sequence $\mathcal{S}_i$ going from position b to position e.

The logical homology matrix is noted $\mathcal{A}$ where $\mathcal{A}_{\mathcal{S}_i,\mathcal{S}_j} \neq 0$ if $\mathcal{S}_i$ and $\mathcal{S}_j$ are homologous and therefore mutually exclusive as queries.

An instance of MPI_MKDOM2 is composed of q processes $\mathcal{P}_1, \cdots, \mathcal{P}_q$. The set of processes is split into a set $\mathcal{W}$ of worker processes $\mathcal{W}_1, \cdots, \mathcal{W}_p$ and a master process $\mathcal{M}_0$.

MPI messages are noted $\langle T \,|\, D \rangle$ where T is the type associated with the message and D are the data sent. The master process sends to the workers either $\langle$ QUERIES $|\, q \rangle$ where q is a list of queries to be processed or $\langle$ UPDATE $|\, u \rangle$ where u is a set of segments to be masked in the database. Workers send back to the master message of type $\langle$ RESULT $|\, q_s, f \rangle$ where q_s is a segment identifying a query and f the domain family created by processing q_s.

2.4.2 Worker algorithm

The worker side of MPI_MKDOM2 presented in Algorithm 3 is fairly straightforward: a worker is an infinite loop waiting for messages from the master. Messages can either be new query batches or database update instructions and the worker acts upon them in the order they are received. In both cases, database updates or query batches, the content of the message is a list of segments. If the message contains QUERIES, the worker fetches the sequence data from the database. The sequence is tested for repeats, and either the sequence or its internal repeat is used to search the database for homologous segments using PSI-BLAST. The resulting matches are parsed to define a set of non overlapping segments. The result is then sent back to the master immediately before processing the next query. If the message is an UPDATE message, all the positions of the segments are masked in the BLAST database and replaced by strings of '*'.

2.4.3 Master algorithm

The main loop of the master process is outlined in Algorithm 6 and goes on until the database is empty. The master process relies on 3 data structures:

segments: a set of segments $\mathcal{S}_i^{b,e}$ representing all the sequence segments that are not yet part of a domain family.

workload: for each worker $\mathcal{W}_k$, *workload*$[\mathcal{W}_k]$ holds the number of queries that have been allocated to this worker and are still to be processed.

result_queue: the role of this data structure is to make sure that results are validated in the same order as the queries were selected. When the master selects a segment $\mathcal{S}_i^{b,e}$ as query,

Algorithm 3: MPI_MKDOM2 worker algorithm.

```
db ← BLAST database created from S
while db not empty do
    Receive ⟨ type | segments ⟩ from M_0
    if type = UPDATE then
        for S_i^{b,e} ∈ segments do
            Mask S_i from position b to position e in db
    else if type = QUERIES then
        for segment S_i^{b,e} ∈ segments do
            if S_i^{b,e} contains a repeat S_i^{b',e'} then
                f ← PSI-BLAST(S_i^{b',e'}, db)
            else
                f ← PSI-BLAST(S_i^{b,e}, db)
            send ⟨ RESULT | S_i^{b,e}, f ⟩ to M_0
```

it pushes $\mathcal{S}_i^{b,e}$ to the back of *result_queue*. When the result for $\mathcal{S}_i^{b,e}$ is received, it takes the place of the query segment in *result_queue*. The results are pulled from the front of the queue for validation.

blocking: for each sequence $\mathcal{S}_i$, $blocking[\mathcal{S}_i]$ records how many homologous sequences are currently selected as query. A segment $\mathcal{S}_i^{b,e}$ is selectable as a query only if $blocking[\mathcal{S}_i] = 0$.

update: a set of segments that contains domains from recently validated families that have yet to be masked in the BLAST databases on the workers.

The master is discharged of all tasks that could be performed by workers in order to keep its load as low as possible. Ultimately, the master process only performs operations that must remain sequential: query selection and result validation.

Query Selection: The query selection algorithm is represented in Algorithm 4. The constant QUERY_THRESHOLD defines the minimum number of queries per batch. For each worker that has fewer than QUERY_THRESHOLD queries to process, a new list of queries is created. Queries are selected by scanning *segments* by order of increasing segment size. A segment $\mathcal{S}_i^{b,e}$ is selected if $blocking[\mathcal{S}_i] = 0$. Once a segment has been selected, it is added to the current list of shortest queries. *blocking* is updated by incrementing by one $blocking[\mathcal{S}_j]$ for all sequences $\mathcal{S}_j$ verifying $\mathcal{A}_{\mathcal{S}_i\mathcal{S}_j} \neq 0$.

Result validation: To validate a family f, we must check that none of its domains overlap with a domain in previously defined families. We use a simpler and equivalent test represented in Algorithm 5, which consists in checking that all the segments within f can be found within *segments*. For each segment $\mathcal{S}_i^{b,e} \in f$ we must find a segment $\mathcal{S}_j^{b',e'} \in segments$ such that $\mathcal{S}_i^{b,e} \subseteq \mathcal{S}_j^{b',e'}$, that is, $\mathcal{S}_j = \mathcal{S}_i$, $b' \leq b$ and $e' \geq e$.
The main loop of the master, illustrated in Algorithm 6, goes through 3 steps:

Algorithm 4: SelectQueries

Input: A reference to *result_queue*
Output: *new_queries*: a table of list of queries

foreach worker $\mathcal{W}_k$ **do**
  if $workload[\mathcal{W}_k] \leq$ QUERY_THRESHOLD **then**
    $new\_queries[\mathcal{W}_k] \leftarrow ()$

foreach $\mathcal{S}_i^{b,e} \in segments$ by increasing order of size **do**
  if $blocking[\mathcal{S}_i] = 0$ **then**
    Select $\mathcal{W}_k$ for which $length(new\_queries[\mathcal{W}_k])$ is minimum
    if length$(new_queries[\mathcal{W}_k])$ = QUERY_THRESHOLD **then break**
    $new\_queries[\mathcal{W}_k] \leftarrow new\_queries[\mathcal{W}_k], \mathcal{S}_i^{b,e}$
    $result\_queue \leftarrow result\_queue, (\mathcal{S}_i^{b,e}, \{\})$
    $workload[\mathcal{W}_k] \leftarrow workload[\mathcal{W}_k] + 1$
    foreach $\mathcal{S}_j, \mathcal{A}_{\mathcal{S}_i,\mathcal{S}_j} \neq 0$ **do**
      $blocking[\mathcal{S}_j] \leftarrow blocking[\mathcal{S}_j] + 1$

return *new_queries*

Algorithm 5: FamilyIsValid

Input: f, a protein domain family
Output: TRUE if the family is valid, FALSE otherwise

foreach segment $\mathcal{S}_i^{b,e} \in f$ **do**
  if $\nexists \mathcal{S}_j^{b',e'} \in segments, \mathcal{S}_i^{b,e} \subseteq \mathcal{S}_j^{b',e'}$ **then**
    return FALSE

return TRUE

1. If *queries_needed* is set to TRUE, the master select new queries following the Algorithm 4 detailed above. The master sends *update* to all the workers and sends new queries. *update* is then reset to an empty set.

2. A result is received from a worker $\mathcal{W}_k$, the workload for $\mathcal{W}_k$ is decremented accordingly. If the worker has less than QUERY_THRESHOLD/2 queries left, *queries_needed* is set to TRUE to trigger the query selection at the beginning of the next iteration. The new family is inserted in *result_queue* in place of the query that created it.

3. All the families available from the front of the queue are validated following the algorithm detailed in Algorithm 5. If a family is valid, its domains are added to *update*, and removed from *segments*, possibly shortening or splitting some segments.

2.4.4 Note on the implementation of MPI_MKDOM2

Implementation of asynchronous communications

The MPI_MKDOM2 algorithm was implemented in C++ and our implementation is, for the most part, faithful to the theoretical algorithm. However some changes were required in order to deal with technical limitations in some MPI implementations. We have attempted to make our implementation independent of specific MPI implementations. It has thus been successfully compiled and tested with various MPI libraries such as `OpenMPI` [20] or `MPICH2` [21]. However, due to technical difficulties, implementation of communications differs from the theoretical algorithm. As explained in Section 1.5 asynchronous communications are a necessity for MPI_MKDOM2. MPI provides the possibility to asynchronously send messages through the `MPI_Isend` function. The `I` in `MPI_Isend` stands for immediate, meaning that the call can return before the receiver has actually received the data. The sender must test that the reception has happened using calls such as `MPI_Test` or `MPI_Wait`. Incomplete non-blocking operations consume resources, for instance each communication in progress requires a handle to identify it. An MPI implementation may have a fixed number of these handles. When it runs out of handles, it will either fail or block new communications until some handles are freed. In the case of MPI_MKDOM2, workers could be blocked during long periods of time processing a query, time during which they are unable to check for incoming messages. The master process must therefore be cautious when sending messages to workers with a large backlog of yet unreceived messages. Implementing a flow control mechanism would have been too cumbersome. Instead we added a second thread to all processes to handle communications. The second thread allows communications to progress even when a worker is otherwise busy for a long time. This additional thread is woken up periodically by the operating system scheduler and explicitly put to sleep after performing the necessary send/receive operations.

Restart mechanism

An MPI_MKDOM2 computation does not have to be performed in a single instance and may be split into several *runs*. MPI_MKDOM2 can be stopped, either intentionally, for instance because allotted time has expired, or unintentionally, for instance after a system or hardware failure. Before launching the next run, domain families that have been validated and recorded to disk by the master in the previous run can be removed from the database. The next run is therefore started on a smaller database which does not contain previously computed families.

Algorithm 6: MPI_MKDOM2 master algorithm

Input: $\mathcal{S}$, a set of protein sequences $\mathcal{S}_1, \ldots, \mathcal{S}_n$
Input: $\mathcal{A}$, a logical matrix of homology between members of $\mathcal{S}$
Input: $\mathcal{W}$, a set of worker processes $\mathcal{W}_1, \ldots, \mathcal{W}_p$
Output: $\mathcal{F}$, a set of protein domain families

foreach sequence $\mathcal{S}_i$ **do**
  $segments \leftarrow segments \cup \mathcal{S}_i^{0,length(\mathcal{S}_i)}$
  $blocking[\mathcal{S}_i] \leftarrow 0$
foreach workers $\mathcal{W}_k$ **do**
  $workload[\mathcal{W}_k] \leftarrow 0$
$result\_queue \leftarrow ()$
$update \leftarrow \{\}$
$queries\_needed \leftarrow$ TRUE
while *segments* not empty **do**
1 **if** $queries\_needed =$ TRUE **then**
    $queries \leftarrow SelectQueries(result\_queue)$
    foreach worker $\mathcal{W}_k$ **do**
      Send $\langle$ UPDATE $|\, update \,\rangle$ to $\mathcal{W}_k$
      if $\exists\, queries[\mathcal{W}_k]$ **then**
        Send $\langle$ QUERIES $|\, queries[\mathcal{W}_k] \,\rangle$ to $\mathcal{W}_k$
    $update \leftarrow \{\}$
    $queries\_needed \leftarrow$ FALSE
2 **wait until** Receive $\langle$ RESULT $|\, \mathcal{S}_q^{b,e}, f \,\rangle$ from any worker $\mathcal{W}_k$
  $result\_queue[\mathcal{S}_q^{b,e}] \leftarrow (\mathcal{S}_q^{b,e}, f)$
  $workload[\mathcal{W}_k] \leftarrow workload[\mathcal{W}_k] - 1$
  if $workload[\mathcal{W}_k] \leq$QUERY_THRESHOLD/2 **then**
    $queries\_needed \leftarrow$ TRUE
3 **while** $result\_queue$ not empty **and** $front(result\_queue)$ has received a result **do**
    $(\mathcal{S}_q^{b,e}, f) \leftarrow front(result\_queue)$
    $result\_queue \leftarrow tail(result\_queue)$
    if $FamilyIsValid(f)$ **then**
      foreach segment $\mathcal{S}_i^{b,e} \in f$ **do**
        $segments \leftarrow segments \backslash \{\mathcal{S}_i^{b,e}\}$
        $update \leftarrow update \cup \{\mathcal{S}_i^{b,e}\}$
      $\mathcal{F} \leftarrow \mathcal{F} \cup \{f\}$
    foreach $\mathcal{S}_j$, $\mathcal{A}_{\mathcal{S}_q,\mathcal{S}_j} \neq 0$ **do**
      $blocking[\mathcal{S}_j] \leftarrow blocking[\mathcal{S}_j] - 1$

However results that were being computed or that were held in queue awaiting validation at the time of the interruption are lost and have to be recomputed.

Memory management on the worker nodes

An MPI_MKDOM2 worker process needs to have access to the sequence data in order to fetch the sequences that have been selected as queries. The PSI-BLAST processes also need to load the database into memory to execute the searches. In order to avoid maintaining two versions of the same database in memory, we take advantage of the `mmap` mechanism that is used by PSI-BLAST to read the target BLAST database. `mmap` is a POSIX system call [62] that maps bytes of a file to addresses in the process' memory space. Data from the file are loaded into memory when the corresponding addresses are accessed. Depending on the memory usage, data can be maintained in memory and may be reaccessed without requiring reading from the file again. `mmap` allows two processes to share the same mapping of a file. MPI_MKDOM2 therefore also uses `mmap` to access the sequence data in the BLAST database without requiring any additional memory. If the machine on which MPI_MKDOM2 runs has enough memory to hold the whole BLAST database in memory, all operations, fetching query sequence data, performing PSI-BLAST searches, and masking domains, can be performed in memory on a common mapping of the database after reading the file once.

2.4.5 Adjacency matrix storage and manipulation

The adjacency matrix in MPI_MKDOM2 represents a large quantity of data, and grows quadratically with respect to the number of sequences. Our first choice to store and access the adjacency matrix was to use an SQLite3 [68] database table. SQLite3 is a lightweight database engine that stores all data in a single portable file. This choice was motivated by the ease of use of SQLite, the indexation mechanism provided by the database engine and its low memory usage. While SQLite3 did provide a convenient and quite efficient way to store and retrieve data, we found that adjacency matrices stored in SQLite3 did not scale up well, to the point that it would become problematic for recent versions of UniProt. For instance for a 210 MB dataset, the adjacency matrix stored as an SQLite3 database already occupies more than 2.3 GB.

To replace SQLite, we defined a format based on 3 binary files: the first file contains the identifiers of all the sequences; the second file contains, for each sequence, a list of its adjacent sequences; and an index file containing for each sequence, a structure with three fields indicating the position of its identifier in the first file, the position of its adjacency list in the second, and the number of adjacencies for this sequence. Figure 2.7 Illustrates the new database format. Identifiers are represented as null terminated text strings. Sequences in the adjacency file are identified by their position in the index file. The three files are loaded into memory by MPI_MKDOM2 using `mmap`, which handles the caching of the data according to available memory. As we select sequences according to their sizes, adjacency data for sequences of similar size are likely to be accessed at approximately the same time by MPI_MKDOM2. We therefore store adjacency information in the different files according to the size of the sequences. Adjacencies stored in this format require 7 to 8 times less disk space than the same adjacency matrix in SQLite3 format.

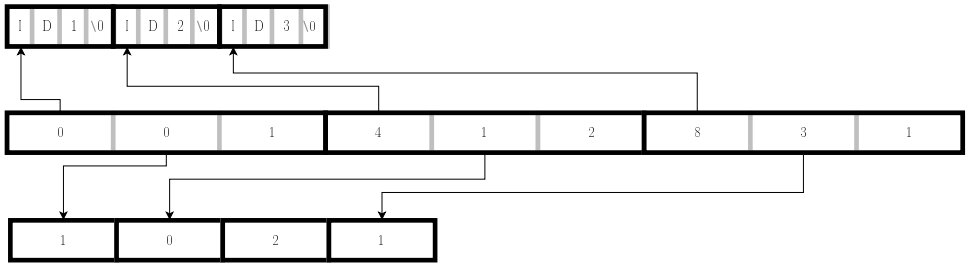

Figure 2.7: Illustration of the adjacency matrix format in MPI_MKDOM2. From top to bottom, arrays represent bytes in: the identifier file, the index file, and the adjacency file. In this example, the matrix stores adjacency relations (ID1, ID2) and (ID2, ID3).

2.5 Experimental evaluation of MPI_MKDOM2's performance

Our goal in designing MPI_MKDOM2 was to develop a method able to produce results close to those of MKDOM2 while being able to process large datasets (on the Gigabyte scale) in a reasonable time-frame. The proximity between the sequential and parallel results is addressed in Chapter 3. In this section we evaluate the second part of this statement, that is, how fast and how efficiently can MPI_MKDOM2 process a dataset. We evaluate two aspects of the runtime performance of MPI_MKDOM2:

Sequential performance: we compare the time required by MKDOM2 and MPI_MKDOM2 in a sequential setup to process the same dataset.

Parallel performance: we assess the evolution of the time needed by MPI_MKDOM2 to process a sequence dataset as a function of the number of workers.

2.5.1 Experimental setup

We based our experiment on the processing of the entry dataset for ProDom 2003.1, here called 'DB'. This 556,964 sequence dataset presents the advantage to be realistic and conveniently sized for our experiments. We created a smaller dataset called DB/8 as a random subset of DB containing 69,621 sequences (an eighth of DB) that could be run sequentially in a reasonable time.

2.5.2 Comparison between MKDOM2 and a sequential run of MPI_MKDOM2

Although MPI_MKDOM2 has been designed as a distributed algorithm, it can be set to run with only one worker selecting queries one by one, thus replicating the behaviour of MKDOM2. In order to test the effect of the sequential optimizations in MPI_MKDOM2 (mainly differences in the way database updates are handled), we processed DB/8 with both MPI_MKDOM2 and MKDOM2. Processing DB/8 using MKDOM2 took 1,420 minutes while it took 1,043 minutes using MPI_MKDOM2. In a sequential execution, MPI_MKDOM2 is more than 25% faster than MKDOM2.

2.5.3 Parallel efficiency

To assess the efficiency of parallelization, we studied the evolution of the time needed by MPI_MKDOM2 to process the DB/8 database as a function of the number of workers. The results are reported in Table 2.1:

- the *wall-clock time* is the time elapsed between the beginning of the processing and its completion;
- the *total compute time* is the cumulative use time of the resources, that is, the wall-clock time multiplied by the number of workers;
- the *speedup* is the acceleration factor achieved with respect to the 1-worker execution;
- the *efficiency* is the total compute time divided by the total compute time of the 1-worker execution.

The speedup, pictured as a function of the number of workers in Figure 2.8 (left-hand side graph), reaches a maximum slightly below 33. In fact, after a quick initial increase, the speedup shows a plateau between 39 and 79 workers (achieving values between 32 and 33), before decreasing. This behaviour is also exemplified by the evolution of the efficiency (see Fig. 2.8, right-hand side graph) which is quite high up to around 31 workers, before dramatically decreasing. We conclude that, for DB/8, our parallelization is quite efficient for up to 31 workers.

Number of workers	Completion time	Total compute time	Speedup	Efficiency
1	14h20'32"	14h20'32"	1.00	1.00
2	7h09'37"	14h19'14"	2.00	1.00
3	5h03'30"	15h10'32"	2.84	0.95
7	2h12'50"	15h29'54"	6.48	0.93
15	1h05'12"	16h18'02"	13.20	0.88
31	0h33'32"	17h19'51"	25.65	0.83
39	0h29'00"	18h51'32"	29.66	0.76
47	0h26'48"	21h00'02"	32.10	0.68
63	0h26'17"	27h36'41"	32.72	0.52
79	0h26'10"	34h28'24"	32.87	0.42
95	0h29'13"	46h16'47"	29.44	0.31
127	0h40'28"	85h40'26"	21.26	0.17

Table 2.1: Summary of parallel performance results.

There are two factors that can negatively influence the speedup as the number of workers increases:

- For a given database size, the average query processing time is the same, yet as there are more workers, the master node must work harder to select queries for all workers. The balance between the time the workers spend processing queries and the time the master needs to process the results and select new queries is crucial.
- The more workers there are, the more queries are processed in parallel and the higher the chances of conflicts. Indeed, the efficiency of our algorithm relies in part on its ability

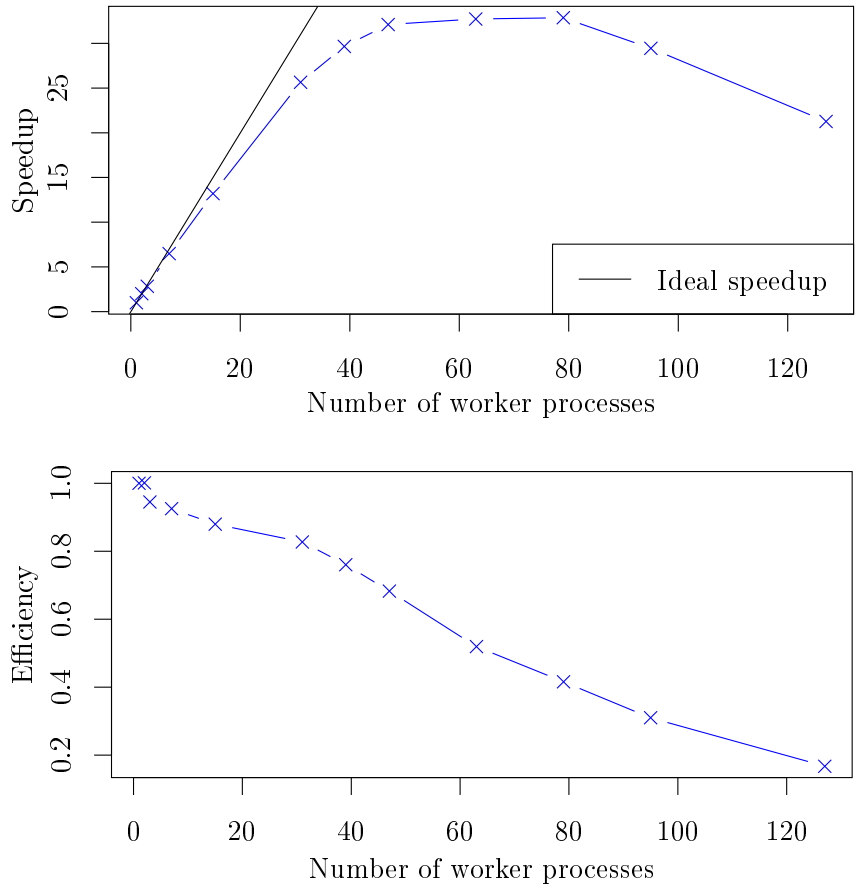

Figure 2.8: Speedup and efficiency as a function of the number of workers for the processing of the DB/8 database.

to predict conflicts between results of different queries. If a significant proportion of the results are discarded, our algorithm will fail to provide a significant improvement in terms of wall-clock time.

The limitation in achievable speedup could therefore be due either to the master process being overloaded by the number of workers to manage, or alternatively by an increase in the frequency of conflicts. Table 2.2 reports the percentage of results whose output had to be discarded as a function of the number of workers: even for a large number of workers, the percentage of conflicts remains low. Our approach for preventing conflicts is thus successful: for instance we observed only 1.59% conflicts when processing DB/8 on 39 workers, *vs.* 25.2% without conflict prediction.

Number of workers	1	2	3	7	15	31
% of conflicts	0.37	0.47	0.55	0.70	0.92	1.35
Number of workers	39	47	63	79	95	127
% of conflicts	1.59	2.18	3.13	3.75	4.22	5.11

Table 2.2: Percentage of queries leading to conflicts as a function of the number of workers, for the processing of DB/8.

Since our conflict mechanism is apparently efficient even with a high number of workers, the decrease of efficiency must be due to the master node. The prospect of trying to process gigabyte-scale datasets given the fact that the speedup reaches a maximum that is less than 33 may look rather grim. However, the larger the database, the larger the achievable speedups. Indeed, with larger databases query processing times will increase, and it should be easier to find non inter-dependent queries. To verify this, we processed the DB database with 39 and 79 workers, which took 57,424 s and 33,403 s respectively, and compared these execution times with those for DB/8. The processing time of DB was divided by 1.72 when going from 39 to 79 workers, whereas that of DB/8 was only divided by 1.11. Therefore the larger the database, the larger the speed-up achievable by MPI_MKDOM2. This was further exemplified during the construction of a new version of ProDom.

2.6 Computing ProDom 2010.1

To demonstrate the ability of MPI_MKDOM2 to efficiently process large datasets, we used MPI_MKDOM2 to compute the family domain clustering on the whole UniProt [57] release 2010_7. After masking low complexity sequence segments with SEG [56], we carried out a global comparison of all sequences using BLAST. This blast search generated 7,875,535,700 matches with an E–value below 10^{-4}. From these matches, we built an adjacency matrix containing 6,664,994,831 relations. Following the methodology used to produce ProDom releases with MKDOM2, UniProt was filtered for partial or preliminary sequences, leaving 6,118,869 sequences or 2,194,382,846 residues to be processed. A first batch of domain families was then defined based on 2,929 families from SCOP [38] release 1.75, leaving 1,896,351,896 residues to be processed by MPI_MKDOM2. This processing was carried out through successive fractional runs on the CINES *jade* cluster and various clusters of GRID'5000 as well as *tomate*, a 16-core shared memory machine, for the last part of the computation. The processing was split into 26 runs. For each run at CINES or on GRID'5000, as many worker processes were started as there are

cores on each node so that each worker process had full access to one core. The master process was treated differently and was placed on a node without any other processes. The main reason for this was to give the master process access to the full physical memory of the machine for handling the adjacency matrix.

Table 2.3 summarizes various aspects of the computation. The complete processing took 12 years and 233 days of CPU time and was carried out in 19 days and 4 hours of wall-clock time. The workers were kept busy 78% of the time on average, and the master process 20%. Although MPI_MKDOM2 was processing thousands of queries in parallel, only 3.5% of the results had to be discarded because of conflicts. MPI_MKDOM2 is therefore very well suited for processing large datasets.

CPU time	110,804 hours (12 years, 233 days)
Wall clock time	460 hours (19 days, 4 hours)
Avg. Worker activity	78.76%
# queries	5,626,277
# invalid queries	198,313 (3.5%)

Table 2.3: Some aspects of the computation of ProDom 2010.1.

Table 2.4 summarizes several aspects for each run:

Platform: platform used for this run.

Wall-clock time: duration of the run in hours.

Number of workers: number of worker processes.

CPU time: the definition of the CPU time depends on the platform used: on *jade* and GRID'5000 clusters, the CPU time is defined as the number of nodes reserved × number of cores per nodes × wall-clock time. On *tomate* which is a shared machine, the CPU time is defined as number of processes × wall-clock time.

Master load: percentage of the time during which the master node is busy, selecting queries or validating results.

Worker load: percentage of the time during which the worker nodes are busy, either processing queries or applying database updates.

Conflicts: percentage of the results that were not valid.

Data processed: the percentage of the initial dataset processed during this run.

Efficiency the efficiency is defined as $\frac{\text{size of the data processed}}{\text{CPU time}}$

Figure 2.9 illustrates the evolution of several aspects of the computation as a function of the wall-clock time. From top to bottom, the different panels represent:

- the evolution of the size of the database;
- the size of the result queue plotted each time a result is tested and therefore removed from the queue;

- the size of the query, plotted with respect to the time of reception of the result for this query;
- the time required to process a query, also plotted with respect the time of reception of the result for this query; .
- a graphical representation of the activity of the different nodes over time.

While the whole processing was performed globally efficiently, some runs were more efficient than others. For some runs such as run 15, the worker load is high yet efficiency is relatively low. This can be attributed to the fact that, at the end of a run, computed results that are queued and awaiting validation are lost. Short runs tend to have a lower efficiency because of this border effect. This is exacerbated as queries get longer because of their larger processing times. Despite the large variations in processing time, MPI_MKDOM2 was still able to find independent queries and maintain a high worker load, even with several thousands of results queued. However, as the number of sequences in the database diminishes, it becomes more and more difficult to efficiently use a large number of workers. This is exemplified by the end of runs 19 to 22, where it became increasingly difficult to find independent queries and consequently, more and more workers were left idle. Therefore when the size of the database decreases, it is necessary to decrease the number of workers to maintain a high efficiency. Run 25 is noteworthy as it exhibits a rare pathological behaviour of MPI_MKDOM2: on very long queries, the internal repeat detection mechanism may fail because of combinatorial explosion of matches between repeats. In MPI_MKDOM2, we reused the internal repeat detection mechanism used by MKDOM2 for simplicity. However for future releases of ProDom it appears important to investigate the possible replacement of the internal repeat detection procedure by a newer, more reliable method.

Run #	**Platform**	**Wall-clock time** (h)	**Number of workers**	**CPU time** (h)	**Master load** (%)	**Worker load** (%)	**Conflicts** (%)	**Data processed** (%)	**Efficiency**
1	CINES	2	32	80	5.74	96.14	0	0.02	3,504
2	CINES	2	152	320	16.67	95.56	1.60	0.21	12,387
3	CINES	2	480	976	14.22	91.83	1.89	0.09	1,697
4	CINES	2	480	976	12.64	85.83	1.54	0.10	1,848
5	CINES	2	480	976	13.01	93.04	0.82	0.04	776
6	CINES	6	480	1,952	22.87	96.32	5.47	2.25	21,293
7	CINES	24	952	23,040	34.93	69.15	12.31	9.72	7,793
8	Grid'5000	5	36	200	4.48	91.89	0.01	0.07	6,058
9	Grid'5000	7	272	1,960	5.89	88.07	2.12	0.75	7,034
10	Grid'5000	8	80	704	2.93	88.22	1.64	0.26	6,858
11	Grid'5000	6	80	528	3.65	96.86	1.79	0.41	14,503
12	Grid'5000	12	144	1,824	4.49	98.07	3.25	2.79	28,209
13	Grid'5000	10	128	1,360	3.85	98.00	6.10	2.21	29,962
14	Grid'5000	9	88	864	3.13	98.08	3.41	1.19	25,470
15	Grid'5000	8	64	4,096	2.70	97.83	2.69	0.53	2,383
16	Grid'5000	8	136	1,152	4.49	97.89	3.62	2.70	43,223
17	Grid'5000	2	136	336	23.79	96.39	5.04	1.23	67,422
18	CINES	24	512	12,480	29.04	99.56	7.20	18.35	27,162
19	CINES	24	512	12,480	50.88	95.46	10.31	18.12	26,825
20	CINES	24	512	12,480	37.79	85.26	4.06	11.52	17,077
21	CINES	24	512	12,480	25.83	72.53	2.72	6.11	9,044
22	CINES	24	512	12,480	20.73	39.26	3.16	2.54	3,760
23	Grid'5000	60	48	3,600	7.26	99.92	0.39	5.68	29,148
24	CINES	24	48	1,344	21.52	98.28	0.57	7.74	106,336
25	tomate	137	15	2,032	18.56	46.86	0.35	4.48	40,728
26	tomate	12	6	84	24.92	39.79	0.02	0.94	205,757

Table 2.4: Some aspects of the runs for the computation of ProDom 2010.1.

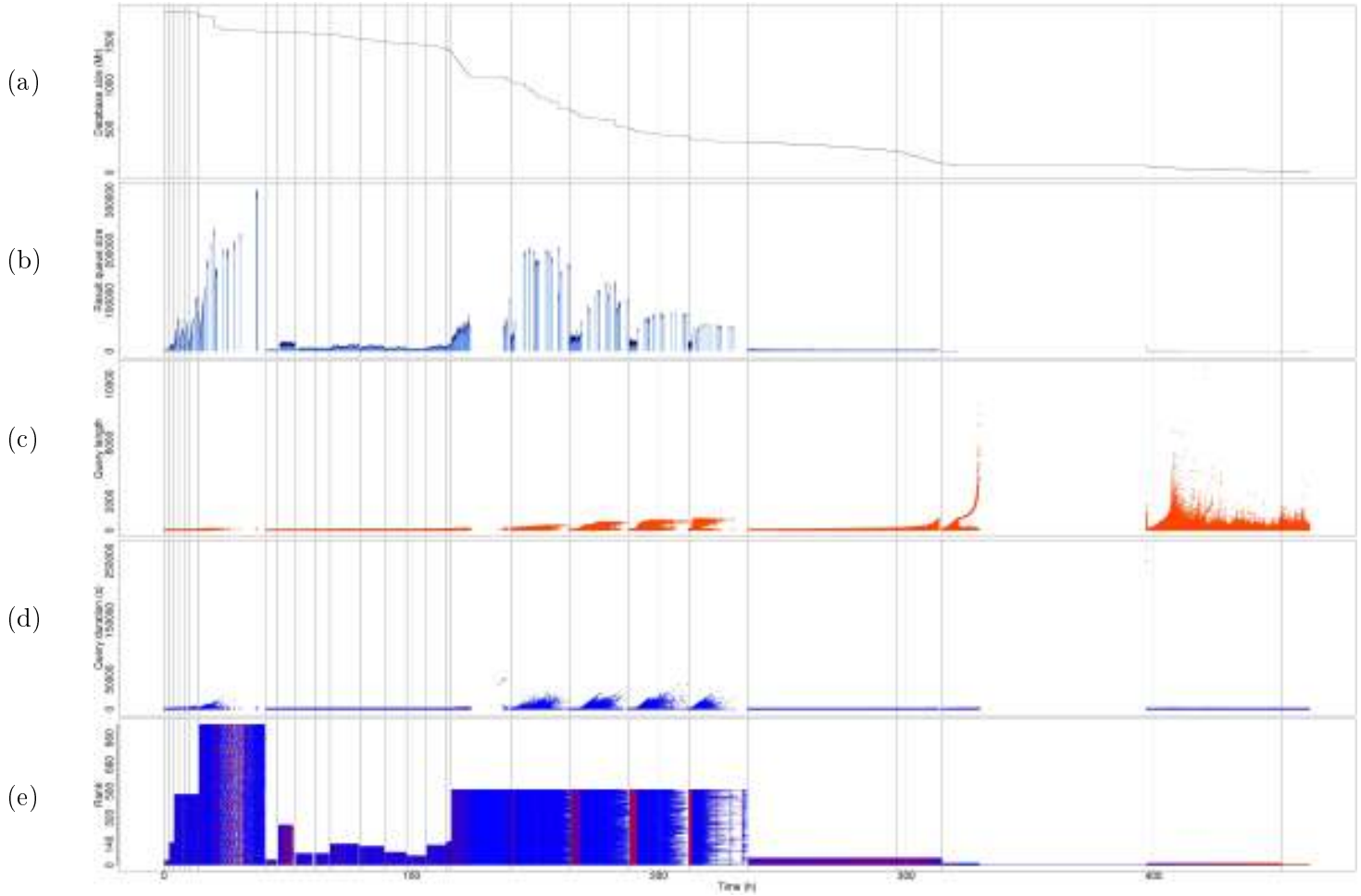

Figure 2.9: Graphical summary of the execution of MPI_MKDOM2 on UniProt 2010_07. From top to bottom: (a) size of database in millions of amino-acids; (b) number of queries considered at a given time; the dark fringe accounts for the queries currently being processed, the light part represents the sequences already processed and awaiting validation; (c) query size in amino-acids; (d) run time of individual queries; (e) worker time-line, a coloured line indicates that the worker is busy either processing queries (blue) or updating the database (red).

Chapter 3

Comparing protein domain clusterings

In the previous chapter, we presented a distributed protein domain clustering algorithm and we showed that this new algorithm could efficiently process gigabyte-scale protein datasets. Yet, we had to cut some corners to allow this speedup: the heuristics behind the MKDOM2 algorithm had to be changed to allow the creation of several protein domain families in parallel. While we designed MPI_MKDOM2 to ensure that the disruption was minimal, it remains to be shown how close sequential and parallel results actually are.

Data clustering is used in many fields of study and the problem of comparing different clusterings has been addressed previously in the literature. In Section 3.1, we present several existing global similarity criteria: the Jaccard index [25], which reflects the proximity of two clusterings; the asymmetrical Wallace indices [53] measuring how well one clustering is conserved in a second clustering; the variation of information [36], a metric measuring the distance between two clusterings.

To our knowledge, these global clustering comparison criteria have never been used for comparing protein domain clusterings. Indeed, variations between two protein domain clusterings may come from the way protein domains are grouped, but domains themselves are not atomic data and may be defined differently in the two clusterings. Consequently, it is not straightforward to apply global clustering comparison criteria to protein domain clusterings. Automatic domain family clustering methods are usually compared to expert defined clusterings in ad-hoc ways by comparing expert defined families to their closest match in the automatic clustering. The closest matching family between two clusterings is hard to define and often requires to experimentally set thresholds and rules. Moreover, the average similarity between families does not necessarily equates to the global correspondence between two clusterings. For these reasons, it is usually hard to globally assess the similarity between two clusterings using these methods. In section 3.2, we present our solution to apply global similarity criteria to protein domain clusterings in a way that takes into account both differences in domain delineation and domain clustering.

Using these global clustering similarity criteria, we study the scalability and stability of the results of MPI_MKDOM2. By processing several queries in parallel MPI_MKDOM2 changes the order in which families are defined and takes into consideration segments which are by construction longer on average than in a sequential processing. We analyze the effects of these changes in Section 3.3.

The database growth that motivated the parallelization of MKDOM2 also has an effect on the results of PSI-BLAST. The sensitivity of a PSI-BLAST search is conditioned by the size of the database and decreases linearly with database growth. These effects on PSI-BLAST results

ultimately impact the clustering produced by MPI_MKDOM2 and we assess this impact in Section 3.4.

3.1 Global clustering comparison criteria

Comparison criteria of clusterings are based on the definition of a clustering as a partition of the inititial dataset, that is, a set of non-empty disjoint subsets that forms a complete cover of the initial set of data points. More formally, given a set of data points $\mathcal{D} = \{d_1, \cdots, d_n\}$, $\mathcal{C} = \{c_1, \cdots, c_p\}$ is a clustering of elements of $\mathcal{D}$ if:

- $\forall c \in \mathcal{C},\ c \neq \emptyset$;
- $\bigcup_{c_i \in \mathcal{C}} c_i = \mathcal{D}$;
- $\forall c_i, c_j \in \mathcal{C},\ c_i \neq c_j \implies c_i \cap c_j = \emptyset$.

3.1.1 Similarity measures based on counting pairs

The intuitive approach to assess the proximity of two clusterings is to count how many elements clustered together in one clustering are still clustered together in the second one. Several measures to compare clusterings are based on this intuition. Given a set of data points $\mathcal{D} = \{d_1, \cdots, d_n\}$, one can define $\frac{n(n-1)}{2}$ distinct pairs $\{d_i, d_j\} \in \mathcal{D}^2, d_i \neq d_j$. Given $\mathcal{C} = \{c_1, \cdots, c_p\}$ and $\mathcal{C}' = \{c'_1, \cdots, c'_q\}$ two clusterings of $\mathcal{D}$, we define π (respectively π') as the sets of pairs of points clustered together in $\mathcal{C}$ (*resp.* $\mathcal{C}'$):

$$\begin{aligned} \pi &= \{\{d_i, d_j\} \,|\, \exists c_k \in \mathcal{C},\ d_i \in c_k \wedge d_j \in c_k\} \\ \pi' &= \{\{d_i, d_j\} \,|\, \exists c'_k \in \mathcal{C}',\ d_i \in c'_k \wedge d_j \in c'_k\} \end{aligned}$$

Jaccard similarity coefficient

The symmetric Jaccard index [25], also known as the Jaccard similarity coefficient, $J(\mathcal{C}, \mathcal{C}')$ [25], measures the fraction of pairs clustered in both $\mathcal{C}$ and $\mathcal{C}'$ relatively to pairs clustered in $\mathcal{C}$ or $\mathcal{C}'$:

$$J(\mathcal{C}, \mathcal{C}') = \frac{|\pi \cap \pi'|}{|\pi \cup \pi'|}.$$

$J(\mathcal{C}, \mathcal{C}') = 1$ when $\mathcal{C} = \mathcal{C}'$, except for $\mathcal{C} = \mathcal{C}' = \{\{d_1\}, \{d_2\} \cdots, \{d_n\}\}$ where it is undefined. Inversely, $J(\mathcal{C}, \mathcal{C}') = 0$ if $\mathcal{C} = \{\{d_1\}, \{d_2\} \cdots, \{d_n\}\}$ and $\mathcal{C}' \neq \mathcal{C}$.

Wallace indices

The asymmetric Wallace W_I [53] (*resp.* W_{II}) index measures the probability that a pair of points clustered together in a given cluster of $\mathcal{C}$ (*resp.* $\mathcal{C}'$) are also grouped in $\mathcal{C}'$ (*resp.* $\mathcal{C}$):

$$W_I(\mathcal{C}, \mathcal{C}') = \frac{|\pi \cap \pi'|}{|\pi|} \text{ and } W_{II}(\mathcal{C}, \mathcal{C}') = \frac{|\pi \cap \pi'|}{|\pi'|}.$$

W_I measures the stability of clustering when going from $\mathcal{C}$ to $\mathcal{C}'$. For instance $W_I = 0.5$ is obtained if a cluster is split into 2 sub-clusters of equal size. W_{II}, on the other hand, measures

how well one clustering is represented by the other. For instance, if a cluster c_1 in $\mathcal{C}$ containing 1,000 data points is split into two clusters c'_1 and c'_2 containing 500 points each in $\mathcal{C}'$, $W_I(c) = 0.5$ and $W_{II} = 1$.

3.1.2 Weighted pair based indices

Because W_I, W_{II}, and J are based on pairs of residues, each cluster contributes according to the square of its size in the final index. In order to counterbalance this effect, we compute weighted indices, where the index of each family is weighted by the number of residues in the family. Weighted Wallace indices W_{w_I} and $W_{w_{II}}$ are defined as:

$$W_{w_I}(\mathcal{C},\mathcal{C}') = \frac{1}{n}\sum_{c_i\in\mathcal{C}} |c_i| \sum_{c'_j\in\mathcal{C}'} \frac{|c_i\cap c'_j|\,(|c_i\cap c'_j|-1)}{|c_i|\,(|c_i|-1)},$$

and

$$W_{w_{II}}(\mathcal{C},\mathcal{C}') = \frac{1}{n}\sum_{c_i\in\mathcal{C}} |c_i| \sum_{c'_j\in\mathcal{C}'} \frac{|c_i\cap c'_j|\,(|c_i\cap c'_j|-1)}{|c'_j|\,(|c'_j|-1)}.$$

Noting that $J(\mathcal{C},\mathcal{C}')$ can be rewritten as follows:

$$\begin{aligned} J(\mathcal{C},\mathcal{C}') &= \frac{|\pi\cap\pi'|}{|\pi\cup\pi'|} \\ &= \frac{|\pi\cap\pi'|}{|\pi|+|\pi'|-|\pi\cap\pi'|} \\ &= \frac{1}{\frac{|\pi|}{|\pi\cap\pi'|}+\frac{|\pi'|}{|\pi\cap\pi'|}-1} \\ J(\mathcal{C},\mathcal{C}') &= \frac{1}{\frac{1}{W_I}+\frac{1}{W_{II}}-1}, \end{aligned}$$

we define $J_w(\mathcal{C},\mathcal{C}')$, a weighted variation of $J(\mathcal{C},\mathcal{C}')$ as:

$$J_w(\mathcal{C},\mathcal{C}') = \frac{1}{\frac{1}{W_{w_I}}+\frac{1}{W_{w_{II}}}-1}.$$

3.1.3 Variation of information distance

More recently, criteria based on information-theoretic concepts have been proposed. These criteria are based on the idea that a clustering $\mathcal{C} = \{c_1,\cdots,c_p\}$ of a set of data points $\mathcal{D} = \{d_1,\cdots,d_n\}$ can be seen as a discrete random variable $X : \mathcal{D} \mapsto \mathcal{C}$: choosing a random point $d_i \in \mathcal{D}$, the value of the random variable corresponds to the cluster $c_i \in \mathcal{C}$ to which d_i belongs. Assuming equiprobability of all the points, we define $p(c_i)$, the probability that a random point d_i belongs to cluster c_i as:

$$p(c_i) = \frac{|c_i|}{n}.$$

The properties characterizing the random variable X can therefore be applied to characterize the clustering $\mathcal{C}$. The "Variation of Information" [36] (VI) is one of the most recent criteria proposed in this category and is based on the notion of conditional entropy of two random variables. The entropy $H(\mathcal{C})$ of $\mathcal{C}$ is defined as the entropy of its random variable:

$$H(\mathcal{C}) = -\sum_{c_i \in \mathcal{C}} p(c_i) \log\left(p(c_i)\right).$$

The entropy characterizes the uncertainty of a random variable. $H(\mathcal{C}) = 0$ if $\mathcal{C} = \{\mathcal{D}\}$ and is maximal if $\mathcal{C} = \{\{d_1\}, \{d_2\}, \cdots, \{d_n\}\}$. Given a second clustering $\mathcal{C}' = \{c'_1, \cdots, c'_q\}$ over $\mathcal{D}$, we can define $p(c_i, c'_j)$, the joint probability of a point $d \in \mathcal{D}$ to be both in the cluster $c_i \in \mathcal{C}$ and in the cluster $c'_j \in \mathcal{C}'$:

$$p(c_i,\, c'_j) = \frac{|c_i \cap c'_j|}{n}.$$

The mutual information $I(\mathcal{C}, \mathcal{C}')$ corresponds to the mutual information of their associated random variables. It measures the information shared by the two clusterings:

$$I(X, Y) = \sum_{c_i \in \mathcal{C}} \sum_{c'_j \in \mathcal{C}'} p(c_i,\, c'_j) \log\left(\frac{p(c_i,\, c'_j)}{p(c_i)p(c'_j)}\right).$$

$I(\mathcal{C}, \mathcal{C}')$ expresses how much knowing the value of one of the variables reduces the entropy (*i.e.*, the uncertainty) of the second one. If $\mathcal{C} = \mathcal{C}'$, $I(\mathcal{C}, \mathcal{C}') = H(\mathcal{C}) = H(\mathcal{C}')$, that is, if both clusters are identical knowing one completely reduces the entropy of the other.

The variation of information between the two clusterings $VI(\mathcal{C}, \mathcal{C}')$ is defined by Meilă [36] as:

$$VI(C, C') = H(C) + H(C') - 2I(C, C').$$

VI is a true metric respecting the triangular inequality for expressing distances between clusterings. $VI(\mathcal{C}, \mathcal{C}') = 0$ if and only if $\mathcal{C} = \mathcal{C}'$. If $\mathcal{C} \neq \mathcal{C}'$ then we have:

$$VI(\mathcal{C}, \mathcal{C}') \geq \frac{2}{n}.$$

The nearest possible neighbor $\mathcal{C}'$ of a clustering $\mathcal{C}$ is obtained when two singleton clusters in $\mathcal{C}$ are merged together in $\mathcal{C}'$ (or inversely, when a 2-point cluster is split). The distance between two clusterings also has an upper bound:

$$VI(\mathcal{C}, \mathcal{C}') \leq \log n.$$

This upper bound is reached with $\mathcal{C} = \{\{d_1\}, \{d_2\}, \cdots, \{d_n\}\}$ and $\mathcal{C}' = \{\mathcal{D}\}$. VI also verifies a stronger bound if $\max(|\mathcal{C}|, |\mathcal{C}'|) \leq \sqrt{n}$:

$$VI(\mathcal{C}, \mathcal{C}') \leq 2 \log\left(\max(|\mathcal{C}|, |\mathcal{C}'|)\right).$$

3.2 Assessing the similarity of protein domain clustering

3.2.1 Comparison by matching segments and families

Authors of automatic domain classification methods usually empirically assess the quality of their results by comparing automatically created families to families extracted from expert sources [23, 42] such as SCOP [38] or Pfam [47]. Each expert family is compared to its closest match in the automatic clustering. The global similarity between the two clusterings is then defined as the average similarity of these one on one comparisons. This approach is however problematic: the closest matching family between two clusterings is hard to define, and the average similarity between families does not necessarily equates to the global correspondence between the two clusterings.

The "best matching family" in our case is ill-defined. Let $\mathcal{C} = \{c_1, \cdots, c_p\}$ be the reference protein domain family clustering, and $\mathcal{C}' = \{c'_1, \cdots, c'_q\}$ an automatically generated set of families. Given a family $c_i \in \mathcal{C}$, there are several ways to define the best matching family $c'_j \in \mathcal{C}'$. A first method may consist in first finding the best match for each domain individually and then choose the family with most domains in common. A second approach would be to find the best match for the whole family in terms of number of residues covered by both families. As illustrated in figure 3.1 both definition are not be equivalent.

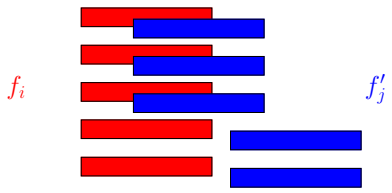

Figure 3.1: Illustration of the differences between matching families based on segments and matching families based on residues. Family f'_j matches the majority of the segments in f_i without matching the majority of its residues.

A second difficulty arises from the fact that the average similarity of families in $\mathcal{C}$ with their best counterparts in $\mathcal{C}'$ does not necessarily equate to the global similarity between $\mathcal{C}$ and $\mathcal{C}'$. As noted by Meilă [35], when comparing clusterings by matching clusters, unmatched parts are unaccounted for. Figure 3.2 illustrates the problem on three families: even though clustering 2 intuitively seems further from the truth than the first one, both would get the same similarity score with a cluster matching criterion.

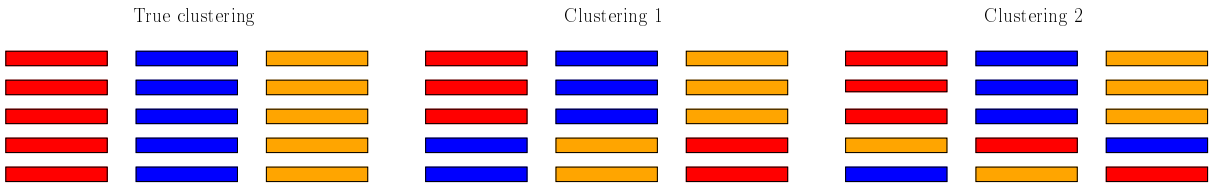

Figure 3.2: Illustration of the problem of cluster matching. Both clusterings 1 and 2 are equivalent if each family is compared to its best match in the true clustering.

All these difficulties make the current evaluation schemes for protein domain family clusterings counter-intuitive, incomplete, and hard to interpret. We propose instead to use residues as

atomic data to apply global clustering comparison criteria.

3.2.2 Applying global criteria to domain families

Global criteria presented in Section 3.1 do not have the same drawbacks as ad-hoc methods. They are able to provide an intuitive reading of the overall similarity between two clusterings. These criteria however cannot be applied directly to domain families.

A protein domain family clustering $\mathcal{F}$ such as those produced by MPI_MKDOM2 do not match the mathematical definition of a partition because the clustered elements, the segments, are not atomic data points. A family of homologous domains $f \in \mathcal{F}$ is a set of segments of the initial sequences that are believed to share a common evolutionary history. Variations between two clusterings can come from the composition of the clusters. For instance, a segment s_1 can be clustered with a second segment s_2 within one clustering but not in the other. Therefore, variations between clusterings can also come from the segments themselves being defined differently by the two clusterings. Comparing clusterings by looking at segments is ill-defined: segments are not atomic data, they can overlap, and there may be more segments in one clustering than in the other. It is therefore difficult to define a bijection between cluster elements in different clusterings without resorting to subjective criteria, thresholds, and rules, that render the analysis harder to interpret.

For the purpose of comparing clusterings, protein domain families can be viewed as sets of residues. In this context, The actual chemical nature of the residues is irrelevant. A residue is uniquely defined by the sequence it belongs to, and its position within this sequence. Residues are atomic data that are unequivocally defined in both clusterings. By basing our analysis on residues, we take into account both the clustering of domains into families and the congruence between domain boundaries.

Domain families defined by MKDOM2 or MPI_MKDOM2 do not cover the full initial dataset. Segments shorter than 20 residues, considered too small to represent an actual domain, are not taken into account. For our analysis however, clusterings are required to be complete partitions of the dataset. In order to have as many residues in each of the partitions, segment extrema that are too short to be considered segments and that do not belong to any family are viewed as singleton families, thus ensuring that both partitions are complete partitions of the initial data set.

Computing global indices on protein domain families

Global criteria are computed based on cardinals of clusters and cluster intersections. Considering a domain family $f \in \mathcal{F}$ as a set of residues, we can define $|f|$, the cardinal of f as:

$$|f| = \sum_{\mathcal{S}_i^{b,e} \in f} (e_i - b_i + 1)$$

Cardinals for all families $f \in \mathcal{F}$ can be linearly computed on a list of segments sorted by family. In order to compute cardinals of intersections between families $f \in \mathcal{F}$ and $f' \in \mathcal{F}'$ we define a partition $\mathcal{P}$ as the set of segments of maximal length on which the both clusterings agree.

Figure 3.3 illustrates the definition of P on an example sequence with two domains defined differently in $\mathcal{F}$ and $\mathcal{F}'$. Sorting segments of $\mathcal{P}$ according to their family $f \in \mathcal{F}$ and $f' \in \mathcal{F}'$ effectively groups segments belonging to the same intersection. The cardinals of all intersections

can then be computed linearly on the sorted list of segments of $\mathcal{P}$, from which we directly derive Wallace indices, Jaccard index and VI distance using the equations given in sections 3.1.2 and 3.1.3.

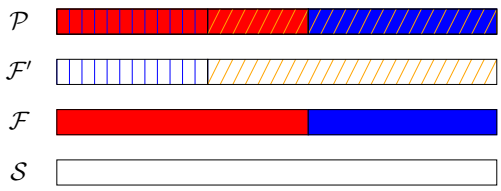

Figure 3.3: Illustration of the relation between $\mathcal{F}$, $\mathcal{F}'$ and $\mathcal{P}$ on a sequence $\mathcal{S}$.

3.3 Measuring the effect of parallelization

In Chapter 2, we tested the parallel efficiency of MPI_MKDOM2 by processing DB/8, a 69,621 sequence database, with an increasing number of workers. We showed that on DB/8, MPI_MKDOM2 was able to efficiently use up to 31 workers, providing a speedup above 25 compared to a 1-worker execution. We designed our algorithm to ensure that the parallelism has as little effect as possible on the results: queries running in parallel are believed to be unrelated and the few remaining conflicts between families are resolved in a conservative way. However, the more workers we use, the more queries are processed in parallel and the further we depart from the sequential behavior. We therefore expect the results to be modified as we increase the number of workers.

To quantify the effect of the parallelization on the results of MPI_MKDOM2, we compared the protein domain clusterings inferred on DB/8 with an increasing number of workers to the clustering produced sequentially on the same database. The top left graph in Figure 3.4 shows the evolution of the W_{w_I} and $W_{w_{II}}$ indices with respect to the sequentially produced clustering as a function of the number of workers. W_{w_I} (resp. W_{wII}) denotes the probability, weighted by family size, that two residues belonging to the same family according to the sequential (resp. parallel) clustering, also belong to the same family according to the parallel (resp. sequential) clustering. W_{w_I} and $W_{w_{II}}$ indices slowly decrease as the number of workers increases, denoting a predictable negative impact of parallelization. However, Wallace indices stay high, above 0.9, as long as fewer than 50 workers are used. This denotes a very good stability of the clustering with regard to parallelization. W_{w_I} indices are larger than $W_{w_{II}}$ indices, meaning that the sequential clustering is better recognized in the parallel clustering, than the opposite is true. The same trends are observable with the VI distance (Figure 3.4, top right graph). VI divergence from the sequentially produced clustering seems to stall at around 0.8, when the maximum distance one may observe between two such clusterings is around 25. One may also note that even the one worker execution results are not identical to the sequential results. This is because in the sequential version, queries are processed and validated one by one, while in the one worker execution, the worker node receives several queries at once (between 10 and 50, depending on the size of the queries).

Because Wallace indices seem to reach a plateau for more than 70 workers, the bottom left graph in Figure 3.4 presents the Wallace indices between clusterings obtained at consecutive points. In other words, in this figure, W_{w_I} (resp. $W_{w_{II}}$) for 15 worker processes is the probability

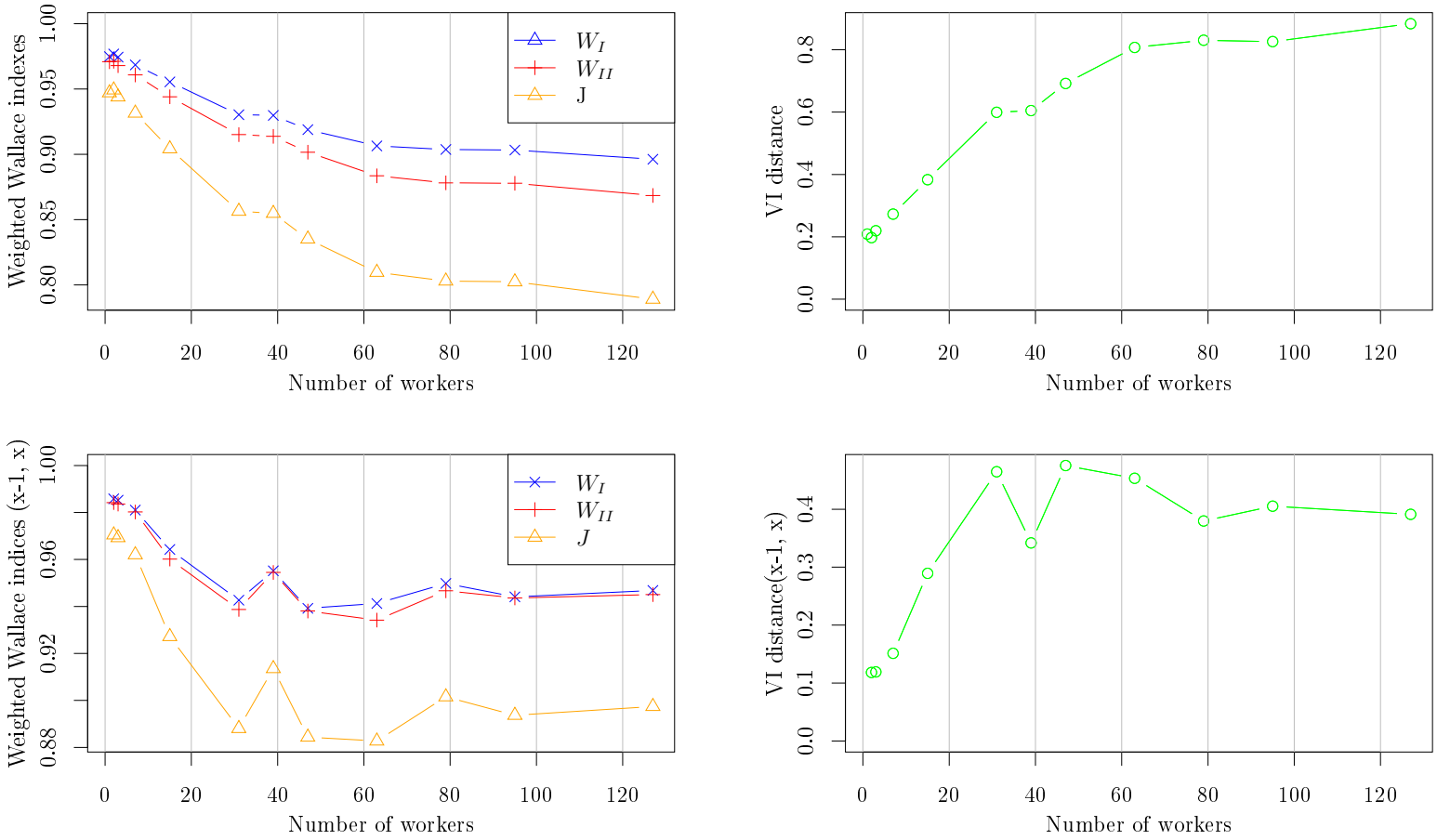

Figure 3.4: Overall assessment of the stability of protein domain families with respect to the number of workers. Top panels compare the clusterings produced in parallel to sequential results, bottom panels compare successive MPI_MKDOM2 clusterings. Left panels shows W_{w_I}, $W_{w_{II}}$, and J_w indices, right panels, the Variation of Information distance.

that two residues belonging to the same family according to MPI_MKDOM2 run on 7 workers (resp. 15 workers) also belong to the same family with 15 workers (resp. 7 workers). Contrary to what the apparent plateau of W_{w_I} and $W_{w_{II}}$ indices could suggest, the structure of the clusterings still evolves slightly when the plateau is reached even if this evolution is quite limited: all Wallace indices are around 0.95. Similarly, the VI distance (Figure 3.4, bottom right graph) between two consecutive clusterings does not tend towards 0. These distances remain however relatively small, around 0.4.

One could expect the chances of conflict between results to be higher with larger families and that, consequently, larger families would be more impacted by parallelization than others. However, we did not find any significant correlation between family size and W_{w_1} even with 127 workers.

Figure 3.5 presents the distribution of the W_{w_I} index with respect to the numbers of domains in families. The left-hand side panels are zooms of right-hand side panels for families containing fewer than 50 domains. Wallace indices do degrade when the number of workers increases. However, even if the Wallace index of a small family can take any value, more than 80% of these families have an index larger than 95%.

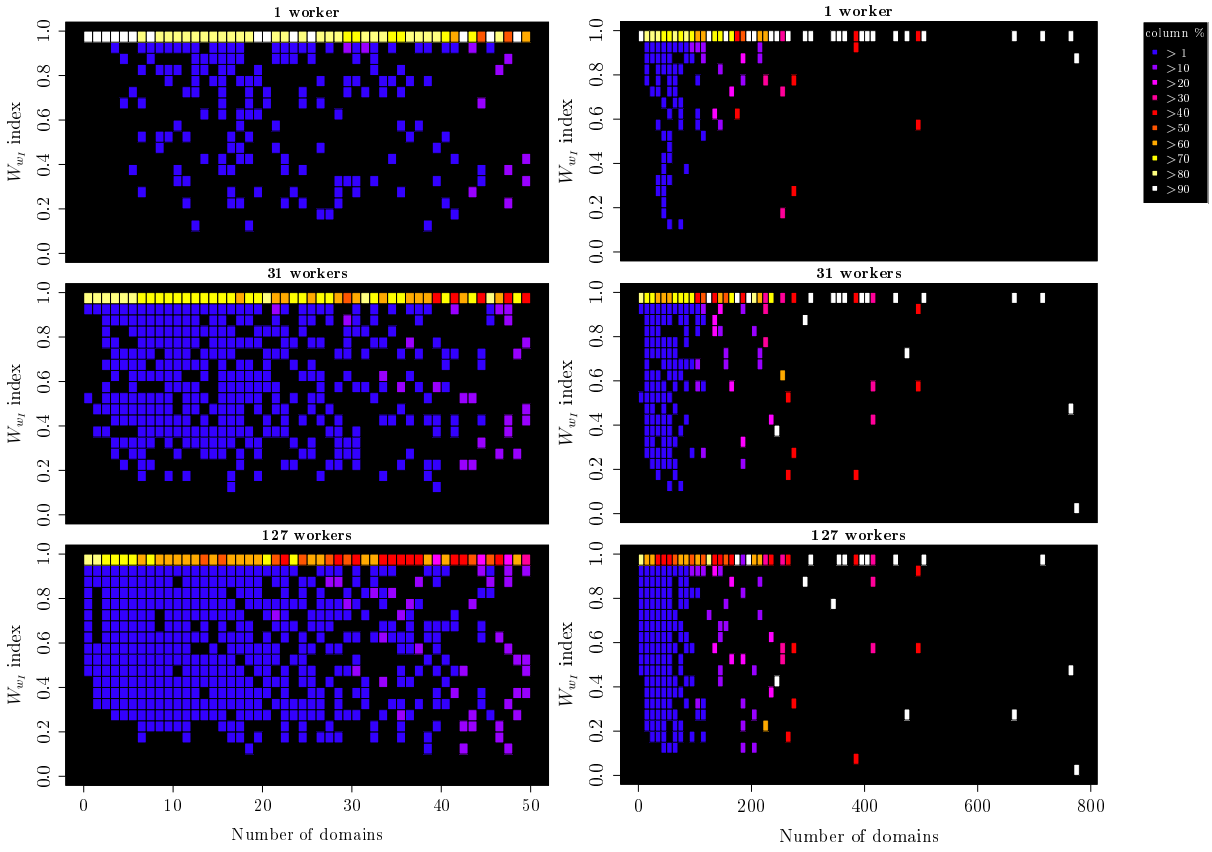

Figure 3.5: Impact of parallelization as a function of family size. Each panel compares the protein domain families built from DB/8 sequentially and in parallel runs respectively with 1, 31, and 127 workers.

From this assessment we conclude that the sequential clustering is essentially preserved in a parallel computation as long as the number of workers is not too large (below 40 workers). The

bound on workers induced by this qualitative constraint —preserving protein domain families— is however less stringent than the bound induced by the computational requirements —the efficiency should stay high. Indeed, as we have seen in Section 2.5, MPI_MKDOM2 starts sharply loosing efficiency above 30 workers while processing DB/8. The number of workers that MPI_MKDOM2 can efficiently use depends on the size of the database (see Section 2.5), but so does the quality of the results: the larger the database, the less likely conflicts become and, as more sequences of similar size are available, the more families we can compute in parallel without departing too much from the sequential heuristics. It therefore does not seem unreasonable to generalize this conclusion and use the computational limitations as a guideline for ensuring the quality of the results.

3.4 Effect of input dataset growth on protein domain clustering

The increase in size of the input database in successive releases of ProDom has been known to cause the subdivision of protein families into subfamilies in successive releases. Although the cause of this problem was known, studying this problem and testing alternative strategies with MKDOM2 on large enough databases would have taken months or years. MPI_MKDOM2 now offers the possibility to quickly carry out experiments on large databases and we can assess the effect of the different solutions using global comparison criteria.

The growth of the input database may modify the composition of domain families in two ways, as illustrated by figure 3.6. A new shorter sequence may become available and split families *vertically*: families that were previously thought to be mono-domain appears to be composed of two different domain families in the light of the new data. This type of modification of existing families is expected and can be beneficial for the quality of the results.

The growth of the input database may also split families *horizontally*. This kind of subdivision is unwanted and is a consequence of the way PSI-BLAST estimates the significance of alignments.

The significance of an alignment between two sequences is assessed by the E–value (see Section 1.2.2). The E–value in BLAST and PSI-BLAST is approximated by:

$$E = \frac{N}{2^{S'}},$$

where N is the number of residues in the target database and S' is the normalized score of the alignment [2]. Because E–values increase linearly with the database size, the sensitivity of homology searches is expected to decrease correspondingly. Matches that, in a previous database had an E–value low enough to be included in a family, may be deemed not significant enough in a larger database.

To alleviate the undesirable effects of database growth, we consider the use of False Discovery Rates (FDR) instead of E–values as homology criterion:

$$\text{FDR} = \frac{\text{E–value}}{\textit{rank}},$$

where *rank* is the rank among the matches sorted by increasing E–values. FDR are expected to be more stable with respect to the variation in size of the database as both E–values and ranks scale linearly with database size.

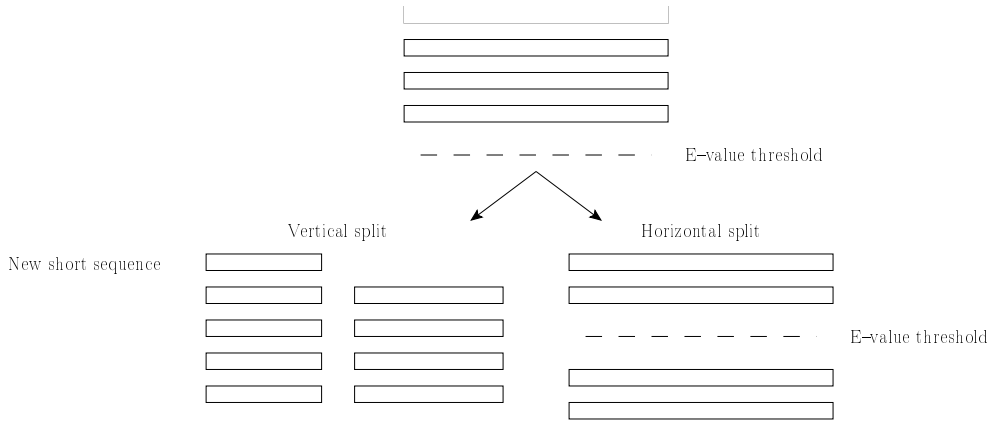

Figure 3.6: Illustration of vertical and horizontal splitting of a family. In a vertical splitting scenario (lower left corner), the addition of a new shorter segment in the database splits all domains in the family in two. In a horizontal splitting scenario (lower right corner), the increased E-value induced by the database growth splits the family into two subfamilies or more.

3.4.1 Assessing the impact of database growth on protein domain clustering

We tested the impact of database growth on domain families by running MPI_MKDOM2 on 4 increasingly larger nested databases. In addition to DB and DB/8 previously used, we define DB/4 and DB/2 which represent respectively a fourth and a half of DB. From DB to DB/8, these databases respectively contain 556,964; 278,482; 139,241; and 69,621 sequences and are nested so that

$$\mathrm{DB}/8 \subset \mathrm{DB}/4 \subset \mathrm{DB}/2 \subset \mathrm{DB}.$$

We ran MPI_MKDOM2 with 39 workers on the four databases using the same E–value criterion as for ProDom releases: a matching segment is considered for inclusion in a family if the E–value of its alignment with the query is below 10^{-6}. We consider the evolution of Wallace indices for the set of nested databases DB/8 to DB. Table 3.1 presents the W_{w_I} and $W_{w_{II}}$ indices for these comparisons. W_{w_I} indices probe the stability of families built from the smaller protein sets when they are embedded in progressively larger sets. As expected, we find that the W_{w_I} index decreases as families are constructed from progressively larger sequence sets when using the E-value criterion (Table 3.1 left hand side).

The right-hand side of Table 3.1 presents the $W_{w_{II}}$ indices. There, the clustering of a large database is compared to the clustering obtained from a subset of this database. The same trends are seen as for the W_{w_I} indices, but these trends are significantly milder as The $W_{w_{II}}$ indices stays nearly stable, around 0.80. This means that the clustering of a large database can be better recognized in the clustering of one of its subsets, than the converse is true. This conveys the idea that what we see in these tables is indeed the effect of the splitting of families induced by the growth of databases.

3.4.2 Effect of FDR on the stability of domain clustering

In order to assess the effect of the use of FDR on the stability of domain clustering, we ran MPI_MKDOM2 on the four nested databases, this time using a FDR threshold of 10^{-6} instead

	W_{w_I}			$W_{w_{II}}$		
	DB/4	DB/2	DB	DB/4	DB/2	DB
DB/8	0.75	0.65	0.56	0.82	0.80	0.80
DB/4		0.72	0.61		0.81	0.78
DB/2			0.70			0.78

Table 3.1: W_{w_I} and $W_{w_{II}}$ indices when comparing the families produced from nested databases while using the E–value.

of E–value as condition for inclusion in a family. We expect the use of the FDR to result in larger families in average, and in a better stability of clustering with respect to database growth. We plot on Figure 3.7, for clusterings with and without FDR, the probability that a family has at least n domains as a function of n. As expected, we find that using FDR instead of E–value does result in a significant increase of the number of large families.

However, as shown by table 3.1, using the FDR does not improve the stability of the results of MPI_MKDOM2, presumably because the diminution of the horizontal splitting is counter-balanced by an increase in vertical splitting of the families. In other words, larger families that are obtained using FDR (Figure 3.7) tend to be subdivided into families with shorter domains.

	W_{w_I}			$W_{w_{II}}$		
	DB/4	DB/2	DB	DB/4	DB/2	DB
DB/8	0.71	0.61	0.54	0.80	0.77	0.77
DB/4		0.69	0.59		0.77	0.75
DB/2			0.66			0.74

Table 3.2: W_{w_I} and $W_{w_{II}}$ indices when comparing the families produced from nested databases while using the FDR.

Figure 3.8 show the average domain length as a function of database size. As the database grows, more information is available and some segments, which were thought to be mono-domain based on a less complete database, are split into two or more shorter segments, pushing the average domain size down. Using FDR however, always leads to shorter domains in average than using E–value.

It had previously been forecast, but not demonstrated, that the growth of databases had a negative impact on the structure of domain families, by splitting domain families into sub-families due to a loss in sensitivity in the homology search. We demonstrated this impact and investigated whether the use of FDR could mitigate it. It appears that using FDR instead of E–value does enable to build larger families, but it also increases domain subdivision. Determining whether this increased subdivision of the domains results in a better inference of the underlying protein organizations would require further study. However, Contrary to our expectation, the use of an FDR criterion instead of the classical E-value criterion does not globally improve the stability of domain families produced by MPI_MKDOM2. In terms of stability of the clusterings produced, using the E-value therefore appears to be a slightly better option than using FDR.

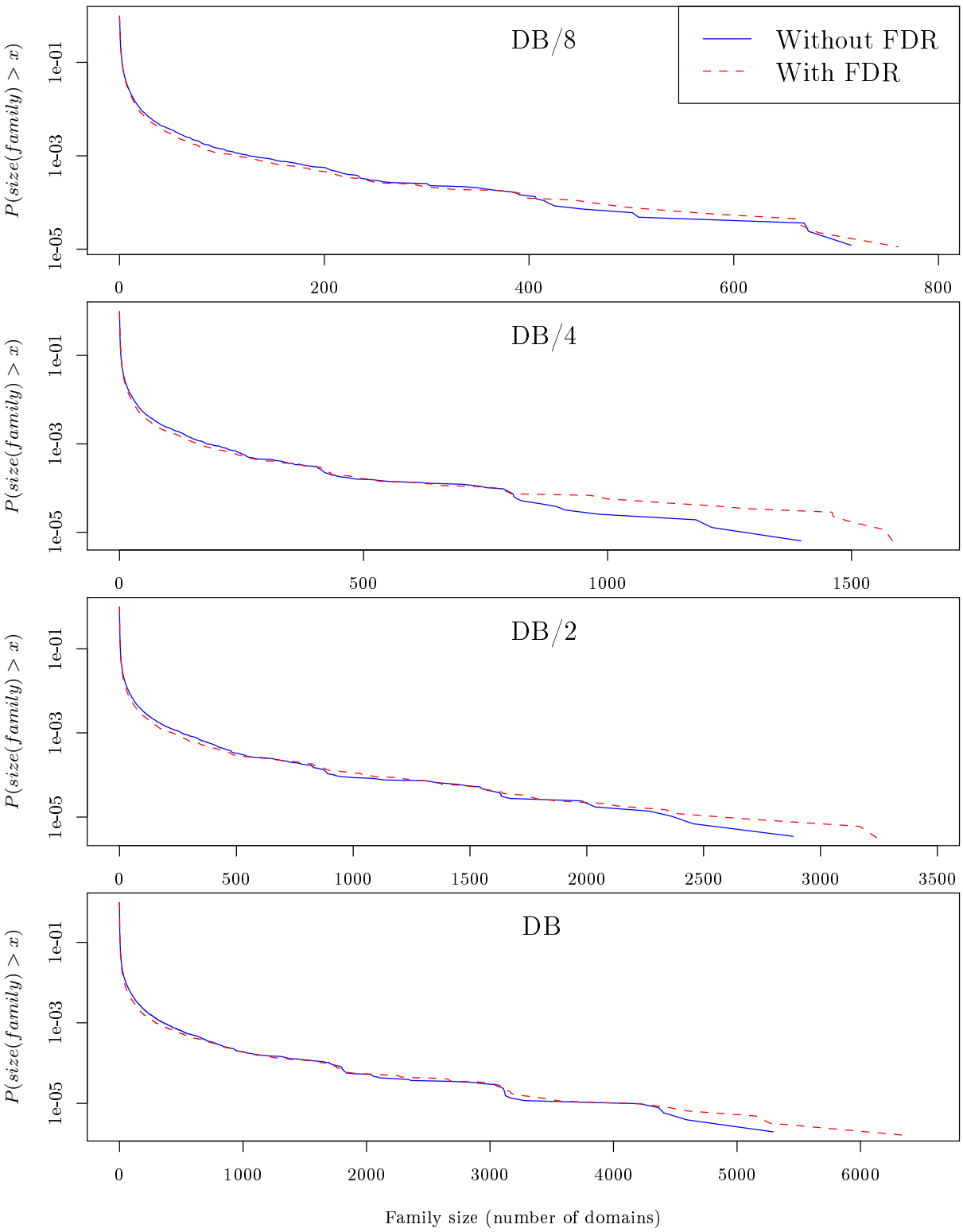

Figure 3.7: Impact of FDR clustering on family size distribution.

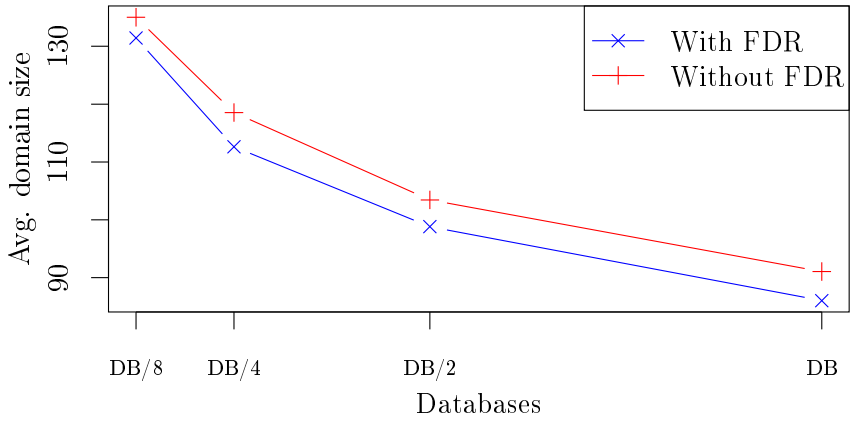

Figure 3.8: Evolution of the average domain size as a function of the database size, with and without FDR.

Chapter 4

Consistent clustering of protein sequences and protein domain families

In the previous chapters, we presented and tested MPI_MKDOM2, a new algorithm allowing the clustering of families of homologous protein domains over a distributed computing platform. We have shown that MPI_MKDOM2 was highly scalable and that it could efficiently process large scale datasets. Yet, as the number of available sequences grows, more and more resources will be necessary to compute new releases of ProDom. As UniProt gets larger, the new sequences that are added into UniProt are more and more likely to be similar to sequences that are already present in the database, because homologous genes may exist in previously sequenced genomes. In order to cope with the rapid increase of sequence data, we envision in this chapter a new algorithm, MPI_MKDOM3, that can infer protein domain families based on families of globally homologous sequences.

This new strategy is also motivated by a bioinformatics methodological problem. In order to understand the genomic mechanisms that drive the evolution of genes and genomes, protein domain classifications are often used jointly with protein sequence classifications. However, computing separately the two types of clusterings often lead to inconsistencies. For instance, the sequence clustering may group sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ into the same family while, according to domain classification, these two sequences do not share the same domain decomposition. We therefore also envision our new algorithm as being part of a pipeline of methods that would build protein domain families based on homologous sequence families, guaranteeing by construction the consistency between the classification of protein sequences and protein domains.

We based our work on the Hogenom[41] database of protein families. We present Hogenom and its current construction method, SiLiX, in section 4.1. It is especially crucial in our case that protein families do not accidentally group non-homologous sequences. We discuss the current developments around Hogenom on this aspect and evaluate how these developments can be applied in the case of MPI_MKDOM3.

In MPI_MKDOM2, homologous domains were detected based on local similarities between sequences. For this new algorithm, we need to find local homologies between protein families instead of protein sequences. There exists several ways to represent and search for local homology between protein sequence families. In Section 4.2 and 4.3, we investigate these different methods in search for the best compromise between speed of execution and accuracy of the results.

We describe in Section 4.4 the new algorithm, MPI_MKDOM3, based on the conclusion of the two previous sections. We summarize our approach, describe how MPI_MKDOM3 relates to MPI_MKDOM2, and how the construction of ProDom can fit in with the construction of

Hogenom.

By compressing the input database, we hope to further reduce the time required to build ProDom releases. However, handling and searching for similarity between protein families is a more complex operation and may require more time than searching for similarity between sequences. In Section 4.5, we assess the performance of the new algorithm at large scale by processing all the sequences available in UniProt as of July 2010.

4.1 Building protein sequence families

The Hogenom database [41] is built by the SiLiX algorithm [37] based on a similarity search of all proteins against all using BLAST, with the BLOSUM62 matrix and an E–value threshold of 10^{-4}. SiLiX defines global homology relations based on the matches found in this similarity search. Two sequences are considered globally homologous if they match over at least 80% of the longest of the two proteins, and if their similarity is over 50%. Two aligned residues are considered similar if their alignment score is positive (see Section 1.2).

Based on the global homology graph, families are built by transitive closure, or single-linkage clustering. For instance, if sequence $\mathcal{S}_1$ is considered homologous to sequence $\mathcal{S}_2$, and $\mathcal{S}_2$ is also homologous to a third sequence $\mathcal{S}_3$, then $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ are grouped into the same family, even if $\mathcal{S}_1$ and $\mathcal{S}_3$ themselves are not homologous.

The constraints put on similarity and alignment length gives most families a good homogeneity. There are however large families for which the transitive closure leads to chimeric families as illustrated by Figure 4.1. These families are composed of sequences that may have nothing in common. In our case, we require that all sequences in a family should have the same domain decomposition, therefore, considering these sequences as one entity would be extremely damaging.

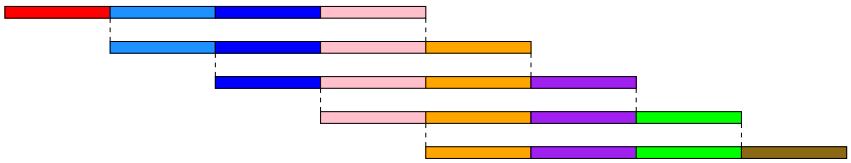

Figure 4.1: Example of a chimeric family built by single linkage clustering. All sequences share 3 out of 4 domains with their neighbor sequences and respect the coverage and similarity threshold. Yet, the top and bottom sequences do not share any similarity.

The current development path chosen by SiLiX authors to address this problem is to use community detection to further split large families according to a modularity criterion with the Newman algorithm [40]. This algorithm creates communities from a graph by maximizing modularity Q defined as:

$$Q = \frac{1}{2m} \sum_{vw} \left(A_{vw} - \frac{k_v k_w}{2m} \right) \delta(c_v, c_w)$$

where m is the number of edges in the graph; $A_{vw} = 1$ if an edge exists between vertices v and w, 0 otherwise; and $\delta(c_v, c_w) = 1$ if vertex v and w are in the same community and 0 otherwise. $\frac{k_v k_w}{2m}$ denotes the probability of existence of an edge between v and w in a random graph of the same size, where k_v and k_w are respectively the degrees of vertices v and w.

Preliminary tests showed that for large families, applying the Newman algorithm is still not sufficient to generate consistent families. This is a known drawback of the modularity maximization approach [15]: as the modularity measure takes into account the size of the graph, the Newman algorithm often fails to detect small communities in large graphs. As an empirical solution to this problem, we apply the Newman algorithm recursively on SiLiX generated families, then on communities until they are sufficiently homogeneous in length (less than 20% difference between the shortest and the longest sequence in the community). This process may lead us to split families more than we would like, and to compress the database less than we could. In our case however, we would rather split too much than too little, as families with the same domain decomposition can be regrouped once the domain decomposition has been established.

We tested the potential compression gains possible by using the SiLiX and recursive Newman approach on the different databases used previously, DB/8 to DB, as well as UniProt 2010_07. Datasets DB/8 to DB have been defined explicitly as subsets. Since UniProt develops mainly by adding new sequences to an existing set, DB which is based on UniProt as of 2003, can also be considered a subset UniProt 2010_07. Figure 4.2 shows the evolution of the number of families or communities as the sequence dataset grows. As expected, there are far fewer families and communities than sequences. On the largest dataset, sequence families represent a 5-fold compression over the number of sequences. While there are more communities than families, applying recursively the Newman algorithm does not completely alleviate the compression. On UniProt 2010_07, there are a third as many communities as there are sequences. Moreover, the ratio of new families or communities to new sequences decreases as the dataset grows. Indeed, as the database grows, new sequences are more likely to find a match in the already existing database. We can therefore expect the compression ratio to increase with future releases of UniProt.

4.2 Detecting local homology between protein families

In the framework we envision, our new algorithm needs to find local homologies between protein families. Protein families are sets of protein sequences that can be aligned globally. The most complete and faithful representation of a protein family is therefore a multiple alignment of all its sequences.

Tools exist such as `alignalign` [30] or `ClustalW` [31] to find the optimal global multiple alignment of two fixed multiple alignments, and also to assess the quality of the resulting multiple alignment by giving it a score such as `norMD` [51]. However to our knowledge there are no existing methods to find local similarity and assess the significance of these similarities directly on multiple alignments. In order to carry out local homology search, multiple alignments must be simplified into different representations. In its simplest form, a multiple alignment can be summarized by a consensus sequence. A consensus sequence contains the most likely residue for each position of the alignment (positions where gaps are most likely are usually removed). Using consensus sequences, one can use classical sequence similarity search algorithms such as BLAST to find local statistically significant homologies between protein families.

The reduction of the multiple alignment to a consensus sequence however represents a strong loss of information. As only the most probable amino acid is conserved at each position, position specific information about the probability of other residues is ignored. Multiple alignments can be more faithfully represented using so called *profile* representations such as PSSMs or HMMs (see Section 1.2.3 and Section 1.2.4). These representations reflect the probability distribution for all 20 amino acids at each column of the alignment,.

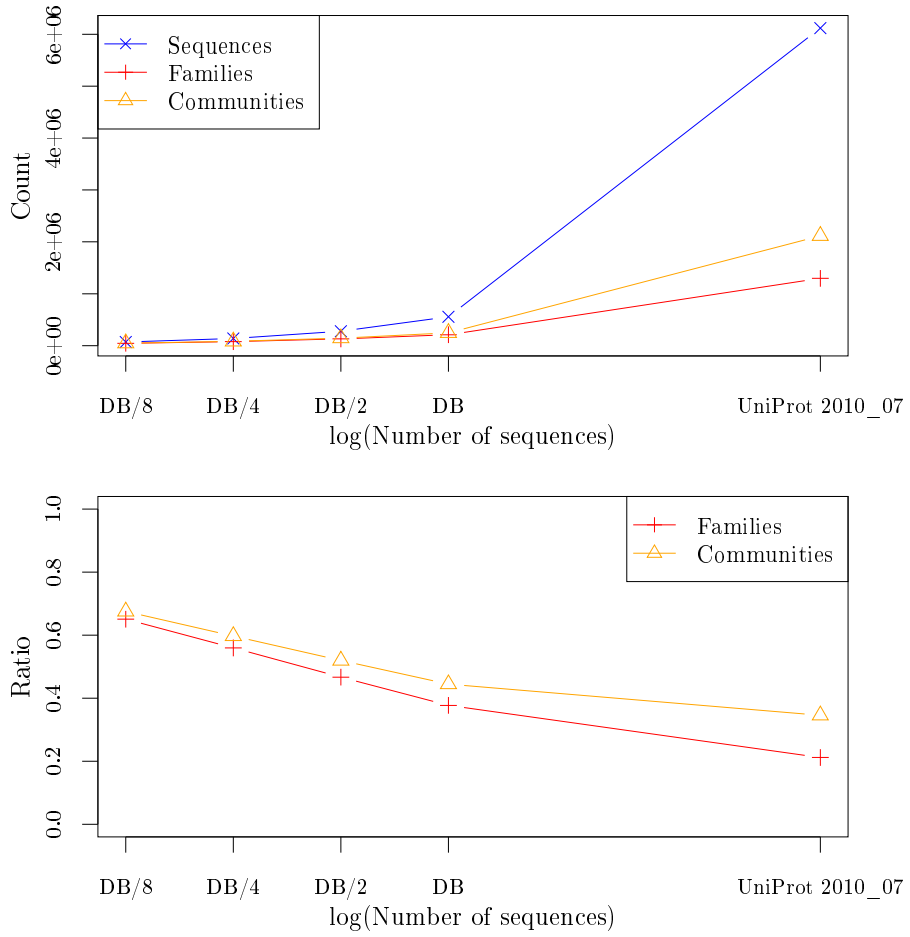

Figure 4.2: Number of protein families and homogeneous communities as a function of the number of sequences in the database (top), and ratio between the number of families or communities and the number of sequences as a function of the number of sequences (bottom).

4.2.1 Profile *vs* profile methods

Several methods, called profile–profile methods, have been developed to allow the detection of statistically significant local homologies between profiles. These methods, such as COMPASS [44], HHsearch [46], or PRC [33], have all been found to be able to detect remote homologies more accurately than sequence–sequence or even profile–sequence methods by their authors [44, 46, 33] or by others [54]. However, these methods are not adapted for the kind of large scale computation we envision.

Profile–profile methods usually use the same dynamic programing techniques as sequence–sequence or profile–sequence methods. They therefore usually have the same worst case complexity in $\mathcal{O}(mn)$ to align two profiles of respective length m and n. The complexity of individual scoring operations for these methods is however higher than for classical sequence–sequence methods such as BLAST, or profile–sequence methods such as PSI-BLAST or HMMER. For instance, PRC defines the score $s(H_1, H_2)$ of aligning two HMMs H_1 and H_2 as the probability of co-emission of the same sequence by the two HMMs, over the probability of emitting the same

sequence according to a null model, summed over all possible sequences:

$$s(H_1, H_2) = log\left(\sum_{\sigma} \frac{P(\sigma|H_1)P(\sigma|H_2)}{P(\sigma|null)}\right)$$

This scoring scheme requires, for each pair of position, $p_i \in H_1$, and $p_j \in H_2$, to consider all the possible pairs between the 20 residues. Comparatively, in a profile–sequence or sequence–sequence approach, scoring the alignment of two positions is a matter of looking up a score or a probability in a table.

More importantly, profile–profile methods do not benefit from the usual alignment seed heuristics (see Section 1.2.2) that is possible when the target database is composed of sequences. When a sequence or a profile is used as query against a target database of sequences with BLAST, PSI-BLAST, or HMMER for instance, a list of short words that can be aligned with the query with a score above a given threshold is generated and the database is scanned to find exact matches for these words and start alignments only where these alignments are likely to be significant. If the target database contains profiles and not sequences, such a heuristics is not possible.

4.2.2 Profile–sequence methods

Profile–profile methods are considered the most accurate way to detect homology between protein families. However, none of the currently available methods are applicable at large scale due to their computational cost. It has recently been suggested that profile–profile searches could be approximated by profile–consensus searches [43]. A profile–consensus search is technically the same as a profile–sequence search, but sequences in the target database are consensus sequences representative of a whole sequence family. This approach in our case seems like a reasonable compromise between accuracy and computational efficiency. There are currently two main tools to carry out profile–sequence alignment: PSI-BLAST and HMMER. We now investigate these two options.

4.3 Comparison between PSI-BLAST and HMMER

Although PSI-BLAST was used to detect homology between sequences in MPI_MKDOM2, it can also be used as a profile–sequence search tool. The first iteration of a PSI-BLAST search (see Section 1.2.3) can be bypassed by providing PSI-BLAST a precomputed profile, or a multiple alignment, as query. Using PSI-BLAST in our case would seem like a convenient choice as only a few modifications to the existing MPI_MKDOM2 algorithm would be required to handle profile–consensus searches. However the recent release of version 3 of HMMER [27] led us to investigate the replacement of PSI-BLAST by HMMER as our similarity search algorithm. Previous releases of HMMER [12] had already been shown to be more accurate but orders of magnitude slower than PSI-BLAST. However in its most recent release HMMER has been updated to incorporate heuristics similar to the seed heuristics used by BLAST. In order to compare the two algorithms, we processed the RefProtDom dataset [18] with PSI-BLAST and HMMER and compared both their efficiency and accuracy. RefProtDom is a dataset designed to benchmark similarity search algorithms. It is composed of a selection of queries and a database containing annotated target sequences. From the annotations, it is easy to determine whether the matches found by similarity search algorithms are true or false positives. Similarly

to PSI-BLAST, HMMER can perform iterative searches starting from a sequence using a tool called jackHMMer [11]. We compared the performances of PSI-BLAST and jackHMMer with two iterations, one to build the PSSM or HMM, a second iteration where the PSSM or HMM is used to search the database.

4.3.1 Speed of execution

Table 4.1 shows the completion time, speedup, and efficiency for PSI-BLAST and HMMER running on 1, 2, 4, and 8 cores. PSI-BLAST on one core is faster than HMMER by a factor of 3.

	PSI-BLAST			HMMER		
Number of cores	Completion time	Speedup	Efficiency	Completion time	Speedup	Efficiency
1	10'14"	1.00	1.00	33'62"	1.00	1.00
2	8'36"	1.21	0.61	17'78"	1.89	0.95
4	7'22"	1.40	0.35	10'06"	3.34	0.84
8	6'90"	1.47	0.18	6'38"	5.27	0.66

Table 4.1: Summary of parallel performance of PSI-BLAST and HMMER on the RefProtDom dataset.

As illustrated by Figure 4.3, HMMER shows a better speedup than PSI-BLAST and when using 4 cores, the running time for HMMER comes close to that of PSI-BLAST. HMMER becomes even faster than PSI-BLAST when using 8 cores. However the efficiency of the execution in this case is quite low, around 0.66.

4.3.2 Result quality

Figure 4.4 shows the ROC curve, that is, the sensitivity as a function of 1-specificity when results are ordered by increasing E–value, for both algorithms. The sensitivity is defined as the fraction of the true positives found by the method with respect to the total number of true matches possible; the specificity is defined as the fraction of true positives among the matches found, 1-specificity is therefore the fraction of false positives among the matches found. A perfect method would find all possible true matches without finding any false positive, and its ROC curve would therefore be a line along the y axis. More generally, the closer to the upper left corner the curve is, the better the method. The ROC curve for HMMER is consistently above that of PSI-BLAST, showing a better accuracy of HMMER in comparison to PSI-BLAST.

From a qualitative point of view, HMMER appears to perform better than PSI-BLAST. From a computational point of view, PSI-BLAST is still faster than HMMER on one processor, but HMMER can, when multithreaded on four cores, become almost as fast as PSI-BLAST while keeping a high enough efficiency. We therefore chose, for MPI_MKDOM3, to replace PSI-BLAST by HMMER similarity searches.

4.4 Description of MPI_MKDOM3

In the previous sections, we established that MPI_MKDOM3 should be part of a pipeline of methods that, altogether, would allow the definition of a consistent classification of proteins and

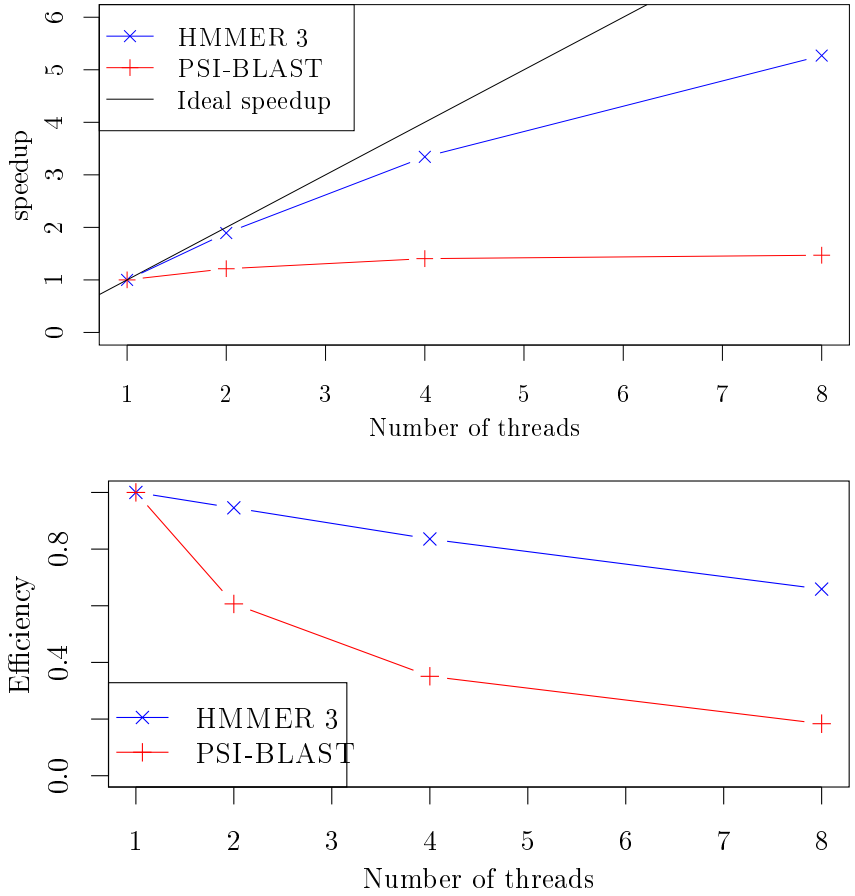

Figure 4.3: Speedup (top graph) and efficiency (bottom graph) of PSI-BLAST and HMMER when processing the RefProtDom dataset as a function of the number of workers.

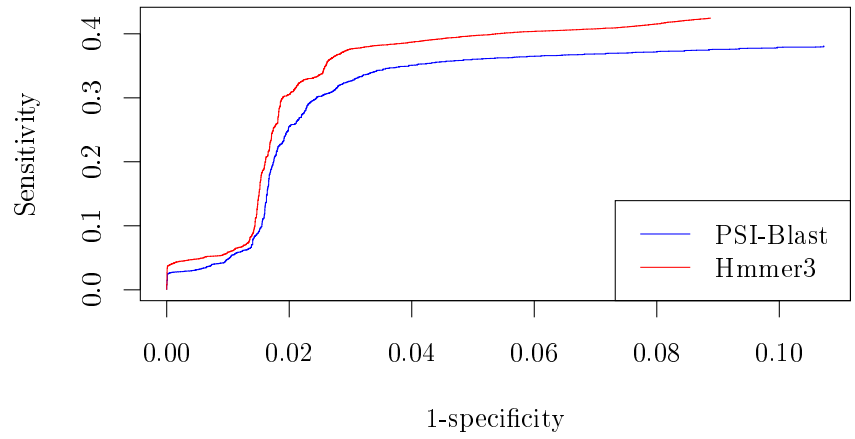

Figure 4.4: ROC curve illustrating the accuracy of PSI-BLAST and HMMER on the RefProtDom dataset.

of protein domain families. This pipeline could be summarized in 8 steps:

1. A set of input sequences is defined, it should be as exhaustive as possible without containing fragments of sequences.
2. A global comparison of all the input sequences is carried out using BLAST.
3. The BLAST matches are parsed with SiLiX to cluster protein sequences into families of homologous sequences.
4. Protein families are split into communities by applying the Newman algorithm recursively.
5. For each protein family we build a multiple alignment and a consensus sequence representative of the family.
6. Protein domain families are computed on protein communities using HMMER to detect local homologies, using HMMs built according to the alignments previously defined, and using consensus sequences as targets.
7. Protein communities that share the same domain decomposition are regrouped to create families of homologous proteins.
8. The protein domains that are defined, on each family, as segments on the consensus sequence are transfered to individual protein sequences.

MPI_MKDOM3, in this scheme, is in charge of carrying out Step 6. MPI_MKDOM3 is based on MPI_MKDOM2. Both algorithms share the same master–worker architecture, the same asynchronous communication methods, and much of the same code. We will now detail the differences between MPI_MKDOM2 and MPI_MKDOM3, on the master and on the worker sides.

4.4.1 MPI_MKDOM3 master process

In MPI_MKDOM2, the master process (see Section 2.4.3) had been defined to be completely ignorant of the actual data, and to only reason in terms of segments, defined by a sequence identifier, a start position and a length (or an end position). Although this design decision was taken for performance reasons when creating MPI_MKDOM2, it appears it also allows us to reuse the same algorithm for the master node of MPI_MKDOM3 as for the master node of MPI_MKDOM2. Indeed, the nature of the similarity search, or of the data used as query is completely irrelevant from the master's point of view. As long as worker nodes accept sequence segments as queries and return sequence segments as results, the master's algorithm does not need to be changed. A slight modification of the code of the master process was made however, to add the possibility to read an additional field in the descriptions of the sequences in the input data file, giving the number of sequences in the family to which a consensus sequence corresponds. This information is used to prioritize larger families when selecting queries in case of an ex-æquo in sequence length.

Similarly to MPI_MKDOM2, MPI_MKDOM3 uses a logical adjacency matrix to avoid processing in parallel queries that are likely to generate overlapping families. The adjacency matrix is constructed from the BLAST results of step 1 like in MPI_MKDOM2: two protein families p_1 and p_2 are defined to be adjacent if at least one sequence $\mathcal{S}_i \in p_1$ is found to be

homologous to a sequence $\mathcal{S}_j \in p_2$ in the initial all against all comparison. Here again, the modification in the construction of the matrix does not require any modification of the master's algorithm.

4.4.2 MPI_MKDOM3 worker process

The worker processes in MPI_MKDOM3 differ on two points from those of MPI_MKDOM2: the query definition and the database masking mechanism.

Query definition

In MPI_MKDOM2, the worker processes received queries defined as sequence segments, that is, a sequence identifier, start position and length (or end position). The worker processes would then fetch the corresponding string of data from the BLAST database to define the query used for the search. In MPI_MKDOM3, the worker processes also receive queries defined as segments of the consensus sequences. But the process to build the query from this information is a little more complicated.

In consensus sequences, positions where gaps are more likely than residues are removed. Consequently, positions on the consensus sequence of a family are not equivalent to positions on the corresponding alignment. From the coordinates of a segment in the consensus sequence of a family, the worker will lookup in a table the corresponding positions on the alignment to only extract the required positions from the alignment. An HMM is then built from these positions to be used as query. For instance figure 4.5 presents an illustration of the correspondence between positions on a multiple alignment and on its associated consensus sequence. Positions 17 to 21 of the multiple alignment are omitted in the consensus sequence. Consequently, if positions 10 to 30 of this consensus sequence were selected as query, the worker should use positions 10 to 35 of the multiple alignment to build the query HMM.

```
                    Multiple alignment
      5      10      15      20      25      30      35      40
S L E T Q Y S T Y D Y L G F N P N F V S D H I P S G E G V V H A E F T N A I G S Y
S L E I Q Y S S Y D Y I G Y N P - - - - - H N N A G E G V V H A A Y S N A L G S Y
S L E V Q Y D S Y D Y I G Y N P - - - - - N N N A G V G V V H A S F A N A L G S Y

                    Consensus sequence
      5      10      15              20      25      30      35
S L E V Q Y S S Y D Y I G Y N P       H N N A G E G V V H A E F A N A L G S Y
```

Figure 4.5: Illustration of the correspondence between positions on the consensus sequence and on the multiple alignment of a family.

Like consensus sequences, HMMs also ignore positions where gaps are most likely. It would therefore seem easier to store pre-built HMMs instead of multiple alignments and thus avoid tedious position mappings. This solution however appeared difficult for two reasons. The first reason is a methodological one. An HMM model is composed of position specific statistical data that are independent of each other. Considering a subset of these data to define a query is therefore possible. But an HMM model also holds statistical parameters [11], used for estimating the E–value of a match, that are globally computed for the whole model. The result of the calibration on the whole HMM will differ from the calibration for a sub-model: to be used correctly, the calibration for the sub-model should be recomputed. The easiest way to do this is

simply to rebuild an HMM based on the corresponding segment of the multiple alignment. The second reason is a practical one. The HMM format defined by HMMER is quite verbose, so storing HMMs instead of multiple alignments would result in a query dataset several orders of magnitude larger, which would be impractical to store and probably inefficient to work with.

Database masking

In MPI_MKDOM2, we optimized the database updates by masking already defined domains with '*' directly in the BLAST formatted database. Contrary to PSI-BLAST, HMMER does not reformat the sequence database, and works directly on the text representation of the database in FASTA format. A protein sequence in FASTA format consists in a single description line, followed by lines of sequence data. The description line is identified by the leading character '>' that should be followed directly after by the identifier for the sequence. The sequence data may be split over several lines and it is recommended that data lines do not exceed 80 characters. Figure 4.7 illustrate a sequence represented in FASTA format. A FASTA file containing multiple sequences consists in the concatenation of several sequences in FASTA format.

```
>seqid1234
MREVKKLYEDEEVEEIALKISDEEIEIVDKKGVVKGKISSKQLAKKALEKKTLKFRIKEA
VDTKITKEDEKSLPSLRELYFDPHLLIYGKRAEEVKGSGYVCPVCGIEIDEYGYCGCGAG
SSLKCENCGAKLSEEEIYVRE
```

Figure 4.6: Example of a sequence in FASTA format.

HMMER unfortunately does not accept '*' as a valid character in its target data file. The masking mechanism we designed for MPI_MKDOM2 therefore cannot be replicated directly in MPI_MKDOM3. In order to update the database file without rewriting the complete datafile at each iteration, we designed a new masking mechanism where masked domains are replaced by artificial sequence description lines. Domains defined by MPI_MKDOM3 are, like those of MKDOM2, at least 20-residue long. If the domain to be masked is embedded within a sequence, the 20 characters corresponding to this residues in the FASTA file are replaced by a string "\n>@XXXXXXXXXXXXXXXX\n" representing a new description line, where @X...X is a newly created artificial sequence identifier. The remaining bits of the sequence now appear to HMMER as two independent sequences. The first character of the new identifier, '@', is a special character common to all artificial sequence identifiers. When a segment with an artificial identifier appears in a search result, the segment is modified to replace the artificial identifier with the actual identifier for the sequence, and correct the position of the segment to reflect its position within the original sequence. If the domain to be masked is longer than 20 residues, the string is left-padded with '\n' characters to reach the desired length. With a minimum domain length of 20 residues, the artificial identifiers can be as large as 16 characters as we have 4 characters which are fixed: two new lines characters, the '>' character marking the start of the sequence, and the '@' character indicating that the identifier is an artificial one. Using case sensitive alpha numeric symbols, this allows more than 4×10^{28} possible artificial identifiers.

As with the previous masking solution, masking operations on the FASTA file allows us to carry out database updates without having to rewrite the whole file. Masking m residues on a database containing n residues only require m write operations instead of $n-m$ with a complete rewrite of the database. Most of the time, $m \ll n$.

Before masking

```
>seqid1234
MREVKKLYEDEEVEEIALKIS DEEIEIVDKKGVVKGKISSK QLAKKALEKKTLKFRIKEA
VDTKITKEDEKSLPSLRELYFDPHLLIYGKRAEEVKGSGYVCPVCGIEIDEYGYCGCGAG
SSLKCENCGAKLSEEEIYVRE
```

After masking

```
>seqid1234
MREVKKLYEDEEVEEIALKIS
>@ipgiwRqyVYvfBVqB
QLAKKALEKKTLKFRIKEA
VDTKITKEDEKSLPSLRELYFDPHLLIYGKRAEEVKGSGYVCPVCGIEIDEYGYCGCGAG
SSLKCENCGAKLSEEEIYVRE
```

Figure 4.7: Illustration of the masking mechanism in MPI_MKDOM3. The highlighted segment in the top sequence is the segment to be masked. The masking mechanism replaces the residue of the masked segment by a new sequence description line and attributes an artificial identifier to the end of the sequence.

4.5 Large scale test of MPI_MKDOM3

As for MPI_MKDOM2, we tested MPI_MKDOM3 at large scale by processing the release 2010_07 of UniProt (see Section 2.6). We reused the result of the filtering and global BLAST comparison carried out in Section 2.6. Running SiLiX on the BLAST matches defined 1,296,176 families. After applying the Newman community detection algorithm recursively, these families were split into 2,116,200 communities. For each community, a multiple alignment of the sequences in this community was created using MAFFT [29], and a consensus sequence was generated by creating an HMM from the multiple alignment and generating the consensus sequence from the HMM using `hmmemit` [11]. From the 7,875,535,700 matches found in the global comparison of the sequences, we defined an adjacency matrix between communities containing 442,127,755 relations. Similarly to MPI_MKDOM2, a first batch of domain families were created based on 2,929 SCOP families from SCOP [38] release 1.75. After this phase, there were 651,828,259 residues still to be processed by MPI_MKDOM3. Compared to the 1,896,351,896 left at this stage when processing the same database with MPI_MKDOM2, this represents almost a 3-fold decrease of the data to be processed due to the compression of individual sequences into families. All this preprocessing represents additional computation compared to MPI_MKDOM2. The longest of these operations, aligning families and creating consensus sequences, took around 5 days on a multi-user 16 core machine. All these operations, including SiLiX, are data-parallel and could easily be parallelized on a larger scale for future computations.

Table 4.2 summarizes various aspects of the computation. The complete processing took 11 years and 306 days of CPU time and was carried out in 4 days and 9 hours of wall-clock time. Despite the fact that the database to be processed was almost 3 times smaller, it took approximately the same amount of CPU time (93% as much) for MPI_MKDOM3 as for MPI_MKDOM2. This reflects the fact that individual similarity search operations are longer with HMMER than they are with PSI-BLAST. However the multi-threading allowed by HMMER significantly reduced the wall-clock time, from 19 days and 4 hours with MPI_MKDOM2 to 4 days and 9

hours with MPI_MKDOM3. The average worker activity was higher in MPI_MKDOM3 than in MPI_MKDOM2. This is probably due to the higher running time of HMMER compared to that of PSI-BLAST: the median query processing time while processing UniProt 2010_07 was 10.3 s with MPI_MKDOM2 while it was 18.5 s with MPI_MKDOM3. It is easier in this case for the master node to select new queries before any worker has finished processing its current queries and therefore, to keep workers as busy as possible. The ratio between the median query processing time of MPI_MKDOM2 and MPI_MKDOM3 observed here on a large database also lead us to think that HMMER multi-threaded performance are not good as those observed on a small test case in Section 4.3.1. There were more than twice as manyit is easier for the master node to select new queries invalid results with MPI_MKDOM3 than with MPI_MKDOM2. This may be caused by the fact that HMMER detects more distant homologies than PSI-BLAST. The homologies between families recorded in the adjacency matrix, as computed with BLAST, may therefore not be as good at predicting conflicts between HMMER results.

CPU time	103,852 hours (11 years, 306 days)
Wall clock time	105 hours (4 days, 9 hours)
Avg. worker activity	94.54%
# queries	3,748,748
# invalid queries	459,226 (12.38%)

Table 4.2: Some aspects of the processing of UniProt 2010_07 with MPI_MKDOM3.

The processing by MPI_MKDOM3 was split into 6 runs. For all the runs on the jade cluster at CINES, we set up 2 workers per node. Each worker had therefore access to 4 cores, HMMER was set up to use the 4 cores available for both HMM construction and similarity search. However contrary to MPI_MKDOM2, we did not give the master process access to a full node. The adjacency matrix on family is much smaller than the adjacency matrix on sequences and could fit in half the memory of a node (15GB or 16GB depending on the node).

Table 4.3 summarize the several aspects for each run:

Platform: platform used for this run.

Wall-clock time: duration of the run in hours.

Number of workers: number of worker processes.

CPU time: the definition of the CPU time depends on the platform used. On *jade* and Grid'5000 clusters, the CPU time is defined as the number of nodes reserved × number of cores per nodes × wall-clock time. On *tomate*, which is a shared machine, the CPU time is defined as number of processes (workers + master) × wall-clock time.

Master load: percentage of time during which the master node is busy, selecting queries or validating results.

Worker load: percentage of time during which worker nodes are busy, either processing queries or applying database updates.

Conflicts: percentage of the results that were not valid.

Data processed: percentage of the initial dataset processed during a given run.

Processing speed: the processing speed is defined as $\frac{\text{size of the data processed}}{\text{CPU time}}$. This measure can be thought of as a measure of the efficiency of processing.

Figure 4.8 illustrates the evolution of several aspects of the computation as a function of wall-clock time. From top to bottom, the different panels represent:

- the evolution of the size of the database;
- the size of the result queue plotted each time a result is tested and therefore removed from the queue;
- the size of the query, plotted with respect to the time of reception of the result for this query;
- the time required to process a query, also plotted with respect the time of reception of the result for this query;
- a graphical representation of the level activity of the different processes over time.

MPI_MKDOM3 runs have globally the same characteristics as for MPI_MKDOM2. The number of results that are queued before validation is however on average larger than for MPI_MKDOM2. This is partly due to the longer processing times of HMMER compared to PSI-BLAST, but this may also be due to a more accute effect of the internal repeat detection mechanism problem. For some reasons that we have not identified, the problems affecting internal repeat detection seem to be more stringent on consensus sequences than on real sequences. Runs 2 and 3 were initially set up to last for 24 hours and had to be cut due to an internal repeat detection that, either crashed a process by filling up the memory, or lasted several hours, blocking the whole process. In order to avoid to be blocked by these problems, we tried, starting from run 3, to replace the adjacency matrix by an empty matrix, effectively allowing any query to be selected. This was justified by the fact that in previous experiments, the number of conflicts would drop as the size of the database would decrease. This induced for runs 3 and 4 a higher rate of conflicts, however for smaller runs on *tomate* the removal of the adjacency matrix does not seem to have had a negative effect on the number of conflicts. We conclude that this solution seems best suited only for accelerating small runs at the end of the processing, and are not a general solution for mitigating internal repeat detection problems. At the end of run 3, after 48 hours of wall-clock time, more than 90% of the initial database was processed, again, the end of the processing was prolonged by internal repeat detection problems. For instance, the last run lasted for 30 hours, of which about 20 hours were spent searching for repeats in 2 sequences.

4.6 Future work

The work presented in this Chapter is a first step towards the integration of HOGENOM and ProDom into a common and consistent source of information that maps both local and global homology relationships between proteins. In order to finalize our new solution we would first wish to compare the results of MPI_MKDOM2 and MPI_MKDOM3 on UniProt 2010_07, using the criterion defined in Chapter 3, and characterize the difference between the results of the two

Run #	**Platform**	**Wall-clock time** (h)	**Number of workers**	**CPU time** (h)	**Master load** (%)	**Worker load** (%)	**Conflicts** (%)	**Data processed** (%)	**Processing speed**
1	CINES	24	511	49,152	25.53	99.74	7.3	48.67	6,454
2	CINES	12	511	24,576	38.87	98.95	4.4	26.12	6,927
3	CINES	12	511	24,576	5.35	82.35	24.30	19.63	5,206
4	CINES	10	127	5,120	12.64	85.83	8.69	3.83	4,880
5	tomate	17	3	68	0.00	99.24	2.36	0.63	59,838
6	tomate	30	11	360	0.00	39.14	3.11	1.12	20,258

Table 4.3: Some aspects of the computation of ProDom 2010.1.

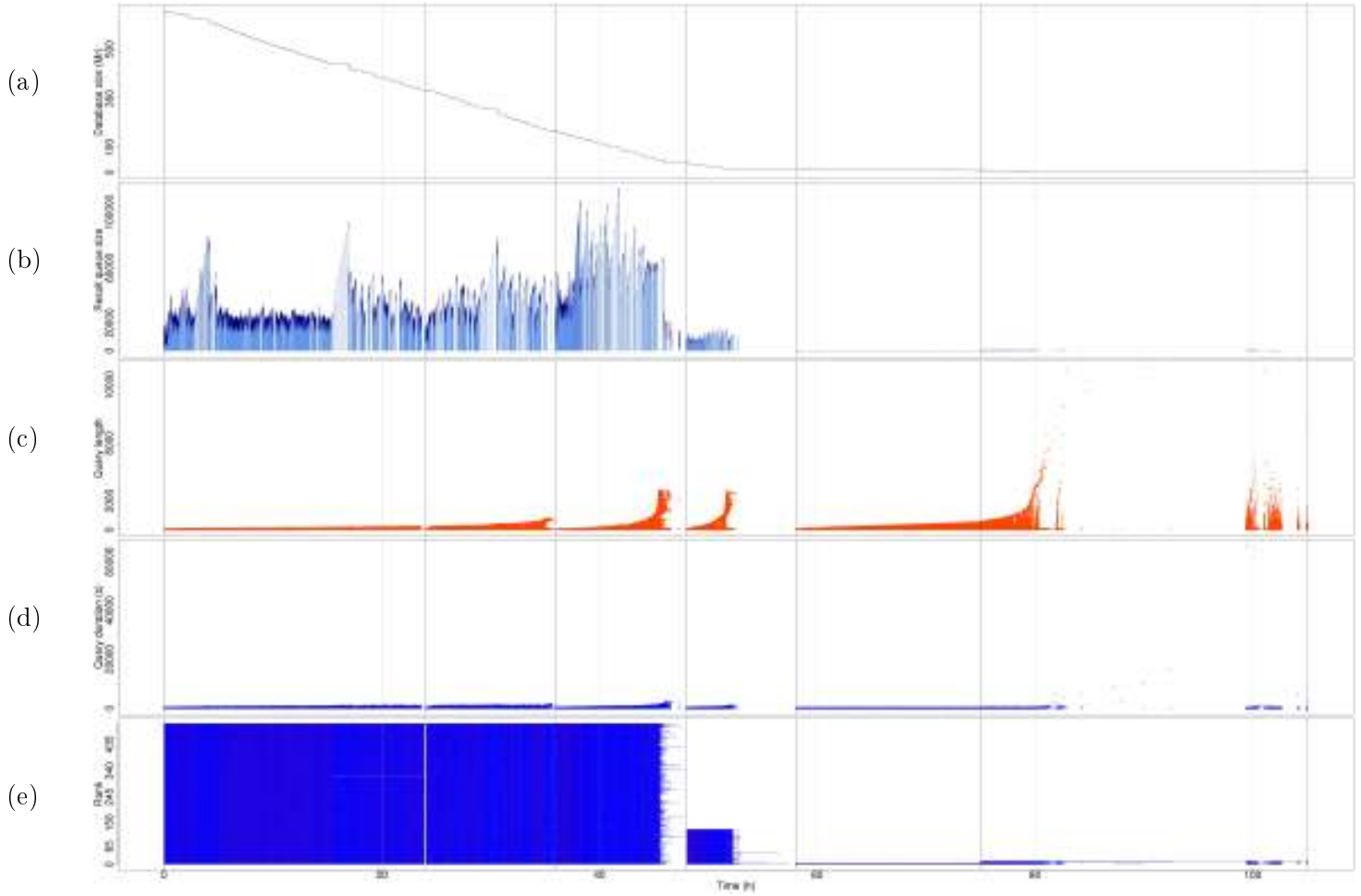

Figure 4.8: MPI_MKDOM3 processing of UniProt 2010_07 monitored over time. From top to bottom: (a) size of database in millions of amino-acids; (b) number of queries considered at a given time; the dark fringe accounts for the queries currently being processed, the light part represents the sequences already processed and awaiting validation; (c) query size in amino-acids; (d) run time of individual queries; (e) worker time-line, a coloured line indicates that the worker is busy either processing queries (blue) or updating the database (red).

methods. A prerequisite for this is to implement the post-processing of MPI_MKDOM3 results. MPI_MKDOM3 outputs domain defined on families' consensus sequences. To complete the process, we would need to first group families that share the same domain decomposition, and then, port the positions of the domains back onto the original individual protein sequences. Similarly, we would like to compare the resulting protein sequence clustering to the clustering that SiLiX alone would produce on the same data.

Before finalizing MPI_MKDOM3, we would also like to investigate a few modifications that may enhance the result quality of MPI_MKDOM3 and at the same time greatly reduce its computational cost:

- The performance of large scale experiments carried out throughout this work were harmed by problems induced by the internal repeat detection procedure. The internal repeat detection program predates even MKDOM2 and has not been updated, or even maintained much since. We chose not to address this problem during the development of MPI_MKDOM2, as we did not know whether they would be reused for MPI_MKDOM3. Instead of correcting the problems in the current procedure, it may be more efficient to simply replace it with a more recent method such as RADAR [22] or Internal Repeat Finder [34]. Additionally we may want to move the internal repeat detection outside of MPI_MKDOM3 and treat it as a preprocessing step. Indeed, as internal repeat detection is done on each sequence independently, it may be more efficient to run it in a data-parallel way prior to MPI_MKDOM3.

- In MPI_MKDOM3 as in MPI_MKDOM2, results are validated in the order queries are selected. When a query takes a long time to process, it blocks the validation of the results of all queries that were launched after it. Long running queries are a problem for MPI_MKDOM3 both in terms of result quality as well as computational efficiency. As new queries must be independent to all the queries that have not yet been validated, it becomes increasingly difficult for the master to select new independent queries. The master is then forced to select longer queries and therefore depart further from the sequential heuristics. MPI_MKDOM3 is fairly resistant to these situations and most of the time manages to keep all workers busy despite the large number of queries awaiting validation. However there are two situations where a large number of queries in the queue can become a problem. The first is when the master node runs out of independent queries, in which case workers are left idle until the blocking query finishes. The second case occurs when the computation is split across several runs: at the end of a run the computed results that have not yet been validated are wasted. The changes in internal repeat detection method may greatly reduce the problem as they are at the root of the pathologically long run times of certain queries. Small modifications to the result ordering policy may also improve the situation. In some instances we may choose to allow results from queries that have been selected after a blocking query to take over. It may also happen that a query (or part of it) has been included in another family before it has been fully processed, in which case it could just be ignored.

The results of these analysis may prompt us to adjust parts of the pipeline before we decide that MPI_MKDOM3 is a good candidate to replace MPI_MKDOM2. MPI_MKDOM3 is therefore not yet production ready, but we are confident that, once completed, the integration of protein sequence and protein domain clustering into a consistent structure will provide new insight on protein evolution.

Concluding summary

Contributions

In this thesis we addressed the problems posed by the exponential increase of the available protein sequences to the delineation and clustering of protein domain families in the context of the ProDom database. Our work in this context can be summarized as follows:

A new method for the construction of ProDom

The ProDom database has been constructed for more than 16 years following a sequential process, but this method is now unable to process current datasets in a reasonable time. We proposed in Chapter 2 a new algorithm, MPI_MKDOM2, able to compute protein domain clusterings on distributed computing platforms. We presented and justified our parallelization approach and explained our solutions to solve or circumvent the problems posed by dependencies between families and the large variations in running times of individual computations. MPI_MKDOM2 has been implemented carefully to minimize resource usage, in particular, memory and communications. We have shown that MPI_MKDOM2 was scalable to large datasets, including current releases of UniProt, which makes MPI_MKDOM2 a good solution for the computation of future releases of ProDom.

The results of the computation carried out on UniProt 2010_07 using MPI_MKDOM2 are currently being post-processed in order to create a new release of ProDom which is scheduled to be released before the end of 2012.

Comparison of protein domain clusterings

Our second contribution, presented in Chapter 3, was motivated by the necessity to compare the results of MPI_MKDOM2 to sequentially computed results in order to assess the impact of parallelization on the clustering. We presented existing criteria to assess the proximity of two data clusterings. We showed that these methods could be adapted to protein domain clusterings, despite the fact that protein domains are not atomic data. Our criterion for comparing domain clusterings thus takes into account both differences in domain delineation and domain clustering.

Using this criterion, we have shown that clusterings computed using MPI_MKDOM2 in parallel were essentially faithful to the results computed sequentially. We also investigated the effect of the data growth on the stability of the clusterings produced by MPI_MKDOM2. We were able to assess this effect and to tested the use of FDR instead of E–value as homology criterion to build families to mitigate it. The use of FDR did counterbalance the loss of sensitivity of the homology search and allowed the construction of larger families. However, it also increased domain subdivision and therefore, was not able to increase the stability of the clustering.

Coherent clustering of protein families and protein domain families

Finally, in Chapter 4, we presented an evolution of the method to address a larger problem, the consistent clustering of protein families and protein domain families. We presented a new algorithm, MPI_MKDOM3, that can delineate domains and cluster protein domain families on families of globally homologous sequences. We outlined how MPI_MKDOM3 could fit in a pipeline of methods that would allow the clustering of homologous domain families, as well as families of globally homologous sequences sharing the same domain architecture. We presented the different solutions we took into consideration and explain our choices based on result quality and computational cost. We demonstrated that MPI_MKDOM3 is applicable at large scale by processing a recent release of UniProt.

Perspective: standing the test of time

The amount of data made available through public sequence databases has grown exponentially for the last decade and nothing indicates that this growth will stop anytime soon. We have shown in this thesis how both MPI_MKDOM2 and MPI_MKDOM3 were able to process current datasets efficiently. However, in a few years, the computing time required to create a new release of ProDom with MPI_MKDOM2 will approach a century. Even if by that time MPI_MKDOM3 has replaced MPI_MKDOM2, and we benefit from the compression of input data into families, the computing power required to compute ProDom will still be considerable. In the meantime, it is predictable that the available computing power will continue to grow. However it is hard to predict how computing platforms will evolve in years to come, and how this computing power will be provided. Cloud computing might become the norm, and large scientific computations could in the future be outsourced to be run on virtual clusters, somewhere on the infrastructure of private providers such as Amazon or Google. Alternatively, with the advent of manycore processors, it might become affordable for a laboratory to maintain its own machine boasting several thousand cores. We believe that MPI is versatile enough to handle this large variety of platforms.

We are also confident that, from a biological point of view, our method will stay relevant in the long run. The current rate of growth of sequence databases forces any manually driven annotation attempts to be partial. Automatic methods will remain the only way to cover the whole protein space. Moreover, in their efforts to cover as much of the protein space as they can, manually curated databases may also introduce a bias toward large, ubiquitous domain families. For a researcher studying evolutionary mechanisms, smaller, more recently evolved domain families are also of interest, and the undiscriminating nature of automatic methods gives a more balanced view of the whole protein space.

Appendix A

Bibliography

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–10, 1990.

[2] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–402, 1997.

[3] CB Anfinsen. The formation and stabilization of protein structure. *Biochemical Journal*, 128(4):737, 1972.

[4] G. Apic, J. Gough, and S.A. Teichmann. Domain combinations in archaeal, eubacterial and eukaryotic proteomes. *Journal of molecular biology*, 310(2):311–325, 2001.

[5] Samuel Blanquart. Extraction et Classification Parallèle des Domaines Protéiques. Mémoire de M2, Université de Rennes-1, 2004.

[6] C. Chothia, J. Gough, C. Vogel, and S.A. Teichmann. Evolution of the protein repertoire. *Science*, 300(5626):1701, 2003.

[7] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[8] A.E. Darling, L. Carey, and W. Feng. The Design, Implementation, and Evaluation of mpiBLAST. *Proceedings of Cluster World Conference & Expo*, 2003.

[9] N. Deshpande, K. J. Addess, W. F. Bluhm, J. C. Merino-Ott, W. Townsend-Merino, Q. Zhang, C. Knezevich, L. Xie, L. Chen, Z. Feng, R. K. Green, J. L. Flippen-Anderson, J. Westbrook, H. M. Berman, and P. E. Bourne. The RCSB Protein Data Bank: a redesigned query system and relational database based on the mmCIF schema. *Nucleic Acids Res*, 33(Database issue):D233–7, 2005.

[10] R. F. Doolittle and P. Bork. Evolutionarily mobile modules in proteins. *Sci Am*, 269(4):50–6, 1993.

[11] S. Eddy. Hmmer user's guide. *Department of Genetics, Washington University School of Medicine*, 3, 2010.

[12] S. R. Eddy. Hmmer: Profile hidden markov models for biological sequence analysis. 2001.

[13] S.R. Eddy. A model of the statistical power of comparative genome sequence analysis. *PLoS Biology*, 3(1):e10, 2005.

[14] Robert D. Finn, Jaina Mistry, John Tate, Penny Coggill, Andreas Heger, Joanne E. Pollington, O. Luke Gavin, Prasad Gunasekaran, Goran Ceric, Kristoffer Forslund, Liisa Holm, Erik L. L. Sonnhammer, Sean R. Eddy, and Alex Bateman. The Pfam protein families database. *Nucleic Acids Research*, 38:D211–D222, 2010.

[15] S. Fortunato and M. Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36, 2007.

[16] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 353–377, 2004.

[17] M.K. Gardner, W. Feng, J. Archuleta, H. Lin, and X. Mal. Parallel genomic sequence-searching on an ad-hoc grid: Experiences, lessons learned, and implications. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 104–es. ACM, 2006.

[18] M.W. Gonzalez and W.R. Pearson. RefProtDom: a protein database with improved domain boundaries and homology relationships. *Bioinformatics*, 26(18):2361, 2010.

[19] J. Gouzy, F. Corpet, and D. Kahn. Whole genome protein domain analysis using a new method for domain clustering. *Comput Chem*, 23(3-4):333–40, 1999.

[20] R. Graham, T. Woodall, and J. Squyres. Open mpi: A flexible high performance mpi. *Parallel Processing and Applied Mathematics*, pages 228–239, 2006.

[21] William Gropp. Mpich2: A new start for MPI implementations. In *Euro PVM/MPI*, volume 2474 of *LNCS*, pages 37–42. Springer-Verlag, 2002.

[22] A. Heger and L. Holm. Rapid automatic detection and alignment of repeats in protein sequences. *Proteins: Structure, Function, and Bioinformatics*, 41(2):224–237, 2000.

[23] A. Heger and L. Holm. Exhaustive enumeration of protein domain families. *J Mol Biol*, 328(3):749–67, 2003.

[24] S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America*, 89(22):10915, 1992.

[25] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura.[A study comparing the distribution of flora in a portion of the Jura Alps]. *Bulletin de la Societe Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[26] F. Jacob. Evolution and tinkering. *Science*, 196(4295):1161–1166, 1977.

[27] L.S. Johnson, S.R. Eddy, and E. Portugaly. Hidden Markov model speed heuristic and iterative HMM search procedure. *BMC bioinformatics*, 11(1):431, 2010.

[28] S. Karlin and S.F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences*, 87(6):2264, 1990.

[29] K. Katoh, K. Kuma, H. Toh, and T. Miyata. Mafft version 5: improvement in accuracy of multiple sequence alignment. *Nucleic acids research*, 33(2):511, 2005.

[30] J. Kececioglu and D. Starrett. Aligning alignments exactly. In *Proceedings of the eighth annual international conference on Resaerch in computational molecular biology*, pages 85–96. ACM, 2004.

[31] MA Larkin, G. Blackshields, NP Brown, R. Chenna, PA McGettigan, H. McWilliam, F. Valentin, IM Wallace, A. Wilm, R. Lopez, et al. Clustalw and clustalx version 2. *Bioinformatics*, 23(21):2947–2948, 2007.

[32] Y.J. Lee, W.F. Hsieh, and C.M. Huang. epsilon-ssvr: A smooth support vector machine for epsilon-insensitive regression. *IEEE Transactions on Knowledge and Data Engineering*, pages 678–685, 2005.

[33] M. Madera. Profile comparer: a program for scoring and aligning profile hidden markov models. *Bioinformatics*, 24(22):2630, 2008.

[34] E.M. Marcotte, M. Pellegrini, T.O. Yeates, and D. Eisenberg. A census of protein repeats. *Journal of molecular biology*, 293(1):151–160, 1999.

[35] M. Meilă. Comparing clusterings. In *Proceedings of the Conference on Computational Learning Theory (COLT)*, 2003.

[36] Marina Meilă. Comparing clusterings—an information based distance. *Journal of Multivariate Analysis*, 98(5):873, 2007.

[37] V. Miele, S. Penel, and L. Duret. Ultra-fast sequence clustering from similarity networks with silix. *BMC bioinformatics*, 12(1):116, 2011.

[38] A. G. Murzin, S. E. Brenner, T. Hubbard, and C. Chothia. SCOP: a structural classification of proteins database for the investigation of sequences and structures. *J Mol Biol*, 247(4):536–40, 1995.

[39] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[40] M.E.J. Newman. Fast algorithm for detecting community structure in networks. *Physical Review E*, 69(6):066133, 2004.

[41] S. Penel, A. M. Arigon, J. F. Dufayard, A. S. Sertier, V. Daubin, L. Duret, M. Gouy, and G. Perrière. Databases of homologous gene families for comparative genomics. *BMC bioinformatics*, 10(Suppl 6):–3, 2009.

[42] E. Portugaly, A. Harel, N. Linial, and M. Linial. EVEREST: automatic identification and classification of protein domains in all protein sequences. *BMC Bioinformatics*, 7:277, 2006.

[43] D. Przybylski and B. Rost. Consensus sequences improve psi-blast through mimicking profile–profile alignments. *Nucleic acids research*, 35(7):2238, 2007.

[44] R. Sadreyev and N. Grishin. Compass: a tool for comparison of multiple protein alignments with assessment of statistical significance. *Journal of molecular biology*, 326(1):317–336, 2003.

[45] TF Smith and MS Waterman. Identification of common molecular subsequences. *J. Mol. Biol*, 147:195–197, 1981.

[46] J. S
"oding. Protein homology detection by hmm–hmm comparison. *Bioinformatics*, 21(7):951, 2005.

[47] E. L. Sonnhammer, S. R. Eddy, E. Birney, A. Bateman, and R. Durbin. Pfam: multiple sequence alignments and HMM-profiles of protein domains. *Nucleic Acids Res*, 26(1):320–2, 1998.

[48] E. L. Sonnhammer and D. Kahn. Modular arrangement of proteins as inferred from analysis of homology. *Protein Sci*, 3(3):482–92, 1994.

[49] T. Sterling, D.J. Becker, D. Savarese, J.E. Dorband, U.A. Ranawake, and C.V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*. Citeseer, 1995.

[50] I.B.M.B.G. Team. Overview of the blue gene/p project. *IBM J. Res. Dev*, 52(1/2), 2008.

[51] J.D. Thompson, F. Plewniak, R. Ripp, J.C. Thierry, and O. Poch. Towards a reliable objective function for multiple sequence alignments1. *Journal of molecular biology*, 314(4):937–951, 2001.

[52] C. UniProt. The universal protein resource (uniprot) in 2010. *Nucleic Acids Research*, 38, 2010.

[53] David L. Wallace. A method for comparing two hierarchical clusterings: comment. *Journal of the American Statistical Association*, 78(383):569–576, 1983.

[54] B. Wallner, H. Fang, T. Ohlson, J. Frey-Sk
"ott, and A. Elofsson. Using evolutionary information for the query and target improves fold recognition. *Proteins: Structure, Function, and Bioinformatics*, 54(2):342–350, 2004.

[55] D. B. Wetlaufer. Nucleation, rapid folding, and globular intrachain regions in proteins. *Proc Natl Acad Sci U S A*, 70(3):697–701, 1973.

[56] J. C. Wootton and S. Federhen. Analysis of compositionally biased regions in sequence databases. *Methods Enzymol*, 266:554–71, 1996.

[57] C. H. Wu, R. Apweiler, A. Bairoch, D. A. Natale, W. C. Barker, B. Boeckmann, S. Ferro, et al. The Universal Protein Resource (UniProt): an expanding universe of protein information. *Nucleic Acids Res*, 34(Database issue):D187–91, 2006.

[58] S. Yooseph, G. Sutton, D. B. Rusch, A. L. Halpern, S. J. Williamson, K. Remington, J. A. Eisen, et al. The Sorcerer II Global Ocean Sampling expedition: expanding the universe of protein families. *PLoS Biol*, 5(3):e16, 2007.

Appendix B

Internet references

[59] Cines. http://www.cines.fr, Retrieved on June, 21 2011.

[60] Grid5000. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home, Retrieved on June, 21 2011.

[61] Jade (sgi altix ice). http://www.cines.fr/spip.php?article518, Retrieved on June, 21 2011.

[62] mmap. http://pubs.opengroup.org/onlinepubs/009695399/functions/mmap.html, Retrieved on June, 21 2011.

[63] Message passing interface. http://www.mcs.anl.gov/research/projects/mpi/, Retrieved on June, 21 2011.

[64] C++ binding issue. http://www.mpi-forum.org/docs/mpi-20-html/node21.htm#Node21, Retrieved on June, 21 2011.

[65] mpiblast | docs / faq. http://www.mpiblast.org/Docs.FAQ.html#other-blast, Retrieved on June, 21 2011.

[66] No, seriously, we've made a release «Xfam Blog. http://xfam.wordpress.com/2011/04/01/no-seriously-weve-made-a-release/, Retrieved on June, 21 2011.

[67] Gip renater website. http://www.renater.fr, Retrieved on June, 21 2011.

[68] Sqlite. http://www.sqlite.org, Retrieved on June, 21 2007.

[69] Uniprotkb/swiss-prot release 2011_06 statistics. http://www.expasy.org/sprot/relnotes/relstat.html, Retrieved on June, 21 2011.

[70] Uniprotkb/trembl release statistics. http://www.ebi.ac.uk/uniprot/TrEMBLstats/, Retrieved on June, 21 2011.