FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Providing Transparent Remote Access to HPC Resources for Graphical Desktop Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Kainrad, BSc
Matrikelnummer 1101537

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Assistant Prof. Dr. Sascha Hunold

Wien, 18. April 2018

Thomas Kainrad	Sascha Hunold

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Providing Transparent Remote Access to HPC Resources for Graphical Desktop Applications

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Thomas Kainrad, BSc
Registration Number 1101537

to the Faculty of Informatics
at the TU Wien
Advisor: Assistant Prof. Dr. Sascha Hunold

Vienna, 18th April, 2018

Thomas Kainrad	Sascha Hunold

Erklärung zur Verfassung der Arbeit

Thomas Kainrad, BSc
Hirschengasse 10/2/12
1060 Wien
Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. April 2018

Thomas Kainrad

Acknowledgements

I want to thank my advisor, Sascha Hunold, for his excellent supervision. His comprehensive feedback led to countless improvements throughout the creation of this thesis. He also kept a clear overview over the various fields that are discussed in this work and helped me to remain focused on the essential questions.

Furthermore, I want to thank all people at Inte:Ligand, who have supported me in developing something meaningful for the life sciences community. Especially, Thierry Langer for giving me the opportunity to work in this field, and Gökhan Ibis for his technical insights.

Also, I thank Bernhard and Fabian for being excellent study and dependable project partners. A heartfelt thank you further goes to Diane for her patience and unwavering excitement about any technical idea that I present to her.

Finally, I want to express gratitude to my parents for their manifold support during my years of study.

Kurzfassung

High-Performance-Computing (HPC)-Ressourcen sind von essentieller Bedeutung für eine Vielzahl von wissenschaftlichen Forschungsbereichen. Zumeist handelt es sich um *Distributed-Memory-Cluster*, die sich aus mehreren Rechenknoten zusammensetzen. Die Arbeit mit diesen Systemen ist oft mühsam, besonders für Wissenschafter ohne formale Informatikausbildung. Sie erfordert die Vorbereitung und Übertragung von Eingangsdaten, das manuelle Sammeln von Ergebnissen, und vor allem Erfahrung im Umgang mit Kommandozeilenanwendungen.

Derzeitige Ambitionen, den Zugang zu HPC-Clustern zu erleichtern, legen den Fokus vor allem auf die Bereitstellung von Web-basierten grafischen Benutzeroberflächen, die es erlauben Rechenjobs an das *Distributed Resource Management System* (DRMS) des jeweiligen Clusters zu schicken. Dies bringt bereits wesentliche Bedienungsvorteile gegenüber dem Benützen der Kommandozeile, aber beseitigt nicht die Notwendigkeit des manuellen Umgangs mit Eingangs- und Ausgangsdaten. Wissenschafter müssen auch mit diesen Oberflächen Daten aus ihren Anwendungen mit grafischer Benutzeroberfläche exportieren, sowie Ergebnisse nach erfolgter Ausführung wieder importieren.

Unser Ansatz ist es, Zugriff auf Remote-Cluster direkt in grafische Desktopanwendungen, die von Wissenschaftern alltäglich verwendet werden, zu integrieren. Die Handhabung von Datenkonvertierung und Netzwerkkommunikation passiert im Hintergrund und ist transparent für den Benutzer, wodurch jegliche Bedienungsschwierigkeiten vermieden werden können. Andererseits muss der Benutzer weiterhin die Kontrolle über alle in Auftrag gegebenen Rechenjobs haben. Dies bedeutet vor allem, dass es möglich sein muss den Fortschritt der Jobs zu überwachen sowie sie gegebenenfalls vorzeitig abzubrechen.

Im Zuge dieser Arbeit wurde eine Serverapplikation entwickelt, die auf HPC-Clustern installiert werden kann und Web-Services zum Starten und Kontrollieren von Rechenjobs zur Verfügung stellt. Die Applikation kann mittels eines automatischen *Deployment*-Prozesses auch mit Clustern in der Amazon Elastic Compute Cloud (EC2) genutzt werden. Zugang zu HPC-Ressourcen wurde mithilfe der entwickelten Web-Services direkt in die LigandScout-Software für molekulare Modellierung integriert. Diese Integration ermöglicht es, direkt aus der grafischen Benutzeroberfläche von LigandScout, große Molekküldatenbanken mittels Virtual Screening auf ihre biologische Aktivität zu untersuchen. Die eigentlichen, aufwendigen Berechnungen finden auf einem entfernten HPC-Cluster statt.

Um dieses Virtual Screening effizient auf Distributed-Memory-Clustern durchführen zu können, entwickelten wir einen Algorithmus zum Aufspalten von Screening-Experimenten in mehrere unabhängige Teilaufgaben. Die dadurch notwendige Aggregation von Fortschrittsdaten sowie der finalen Ergebnisdaten passiert ebenfalls transparent für den Benutzer.

Unsere Evaluierung beschäftigt sich hauptsächlich mit der Skalierbarkeit der entwickelten Software und analysiert weiters die Auswirkungen des Algorithmus zur Aufspaltung von Screening-Experimenten auf verschiedenen Clustern. Es wird gezeigt, dass unser Ansatz beinahe linear mit der Anzahl an Cluster-Knoten skaliert. Der Mehraufwand durch Netzwerkkommunikation sowie durch automatisches Aufspalten von Jobs ist insbesondere bei großen Screening Experimenten vernachlässigbar.

Abstract

High Performance Computing (HPC) resources play a major role in facilitating scientific research. Mostly, these resources are present in the form of distributed-memory clusters consisting of multiple compute nodes. Working with these clusters is often cumbersome, especially for researches without formal background in computer science. It requires preparation and transfer of the input data, manual gathering of results, and command-line expertise.

Current approaches for improving accessibility to remote HPC clusters focus on providing web-based graphical front-ends, which allow to submit jobs to the distributed resource management system (DRMS) running on the cluster. This comes with significant usability benefits over command-line usage, but does not circumvent the need for manually handling the input and output files. Scientists still have to export data from their graphical user interface (GUI) applications and re-import results later after the remote jobs are completed.

We propose a different solution, namely to provide remote access to HPC resources directly within the graphical desktop applications, that scientists use in their day-to-day work. By handling necessary data conversion and network communication transparently to the user, this approach allows to completely evade any HPC usability barriers. At the same time, the user has to stay in control of the executed computational jobs, which requires access to DRMS features such as monitoring and cancelling running or pending jobs.

In the course of this work, we have developed a server application that can be deployed on HPC clusters and which provides web-services for submitting and controlling computational jobs. Apart from on-site private clusters, the server software can also be used with clusters in the Amazon Elastic Compute Cloud (EC2) via an automated deployment process. As a practical use case, we have integrated transparent access to HPC resources into the LigandScout molecular design software through background invocations of the developed web-services. This solution allows to virtually screen large compound libraries on remote HPC clusters from within the LigandScout GUI. In order to run the screening algorithm on distributed-memory clusters, screening jobs are automatically split into multiple sub-jobs that can run on individual nodes. Progress and result aggregation is once again transparent to the user.

Our evaluation focuses on the scaling capabilities of the developed software and explores the implications of the implemented job-splitting algorithm on different clusters. It is shown that our approach scales almost linearly with the number of cluster nodes. The introduced overhead through network communication and job-splitting is negligible when the screening experiment is sufficiently large.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

High Performance Computing (HPC) has long been a driving factor for scientific progress and continues to increase its relevance, as service-oriented cloud computing products enable more users to gain access to large computing resources [1]. The benefits of HPC infrastructures, such as compute clusters, over traditional consumer hardware are manifold. The most obvious advantage are shorter execution times, especially for tasks that can be parallelized to a high degree. Another advantage is a decreased load on client machines such as desktop computers or notebooks, which are generally not built for long compute-intensive tasks.

However, working with remote HPC clusters is often times cumbersome for different reasons. It requires preparation and transfer of the input data, manual gathering of results, and most importantly technical knowledge. Scientists therefore often have to rely on the usability of graphical user interface (GUI) applications installed on their own machines over the superior computational capabilities of remote high performance resources. This dependence unnecessarily lengthens the time needed for executing tasks and causes an under-utilization of existing expensive computing hardware.

Naturally, many concepts and approaches have been developed for making HPC resources more accessible. Especially, scientists with limited information technology background rely on these existing solutions, which are largely focused on providing web-based interfaces for cluster resource managers [2, 3, 4]. While this approach circumvents the need for specialized technical knowledge, such as command-line expertise, it still requires manual steps regarding file upload, results download, and the filling of forms. Furthermore, systems striving to be versatile enough to enable access to a wide spectrum of computational pipelines can seldom provide detailed progress information and intermediate results, as this information is specific to individual tasks.

Providing capabilities to access remote HPC resources directly from within GUI applications has tremendous usability benefits. Being able to execute tasks on data already loaded within an application circumvents the need for tedious importing and exporting steps, which may even require manual file conversions. There are various questions as to how such an integration would ideally be implemented, especially when dealing with already existing programs that have not been built to work with remote HPC resources. A robust way for integrating remote computation services into graphical client applications is therefore highly valuable. These remote computation services must be deployable on HPC clusters and able to provide progress information as well as intermediate results. A further need is to minimize data transfers that could potentially exhaust the network bandwidth.

One particular software that offers a GUI and operates in the scientific domain of pharmacy, chemistry, and biology is LigandScout [5]. It is a tool for modeling 3D molecule-protein interactions as pharmacophore models and subsequently for screening large molecule databases against these models, a technique commonly known as virtual screening (VS). The aim is to obtain early-stage drug candidates.

This VS process can easily be done in parallel and can benefit greatly from large computing resources. Currently, those screening jobs are either executed on HPC resources via command-line applications or most often just directly using the graphical desktop application on consumer hardware. Enabling scientific users to harness powerful computing clusters without ever having to worry about command-line parameters, or how to provide their data to remote machines could vastly improve frequent drug discovery workflows.

This work aims to explore the theoretical background for integrating transparent remote access to HPC resources into graphical desktop applications as well as to provide a practical solution for the LigandScout application and its VS algorithm. An additional focus is further placed on cloud computing and the associated scalability and elasticity benefits. The goal is to facilitate automated cloud deployments and in this way also enable organizations without access to private HPC clusters to use the developed solution.

1.2 Structure of the Thesis

The structure of the thesis is as follows: After this introduction, Chapter 2 introduces the most important concepts of HPC, especially regarding cluster computing, user-friendly access to HPC resources, and HPC in the cloud. The background chapter concludes with a description of the LigandScout software and an introduction to the VS concept. Chapter 3 summarizes the efforts of existing solutions and outlines the remaining research questions that this thesis addresses. Chapter 4 then presents the abstract architecture and design decisions involved in the development of the practical solution and Chapter 5 covers the concrete implementation. Experimental results are shown in Chapter 6, before Chapter 7 concludes the thesis with a final résumé and an outlook on potential future work.

CHAPTER 2

Background

This chapter introduces various concepts and technologies necessary for a profound understanding of the later parts of this thesis. In particular, this includes topics related to HPC with a special focus on cluster and cloud computing. Moreover, the LigandScout molecular design software is described, because it serves as an exemplary graphical desktop application for the practical part of this work. Consequently, also the VS process is presented in detail. The chapter further establishes a taxonomical foundation that can later be referenced.

2.1 Parallel Computing

HPC is a broad term, generally used to describe computing environments that make use of supercomputers or otherwise powerful computing resources. In all cases, HPC is dependent on a high degree of parallelism, which is why the terms HPC and parallel computing are strongly connected. In general, parallel computing describes the solving of a problem by multiple processors in a cooperative manner, an exercise that is now done by a large majority of modern computers also apart from HPC use cases. The two basic categories of parallel computing are explained below.

2.1.1 Shared-memory Computing

Shared-memory computers, as the name suggests, work on a common physical address space [6, p. 100]. This behaviour is largely transparent to the programmer, as all threads have access to the same data. Nevertheless, there are different types of shared-memory systems that application developers need to be aware of because of their significantly different performance characteristics. Figure 2.1 shows the basic architecture of shared-memory computers. Concretely, the simplified programmer's view of a shared-memory computer containing four processor cores with two data cache levels (L1D & L2) each is

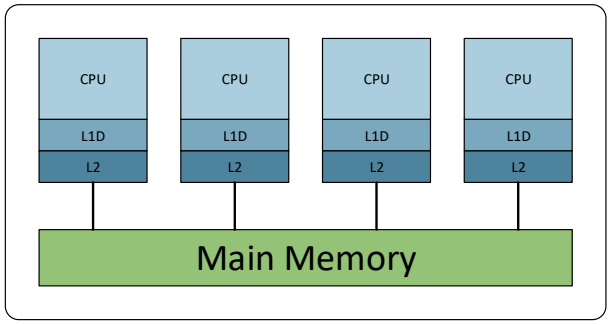

Figure 2.1: Shared-memory parallel computing architecture.

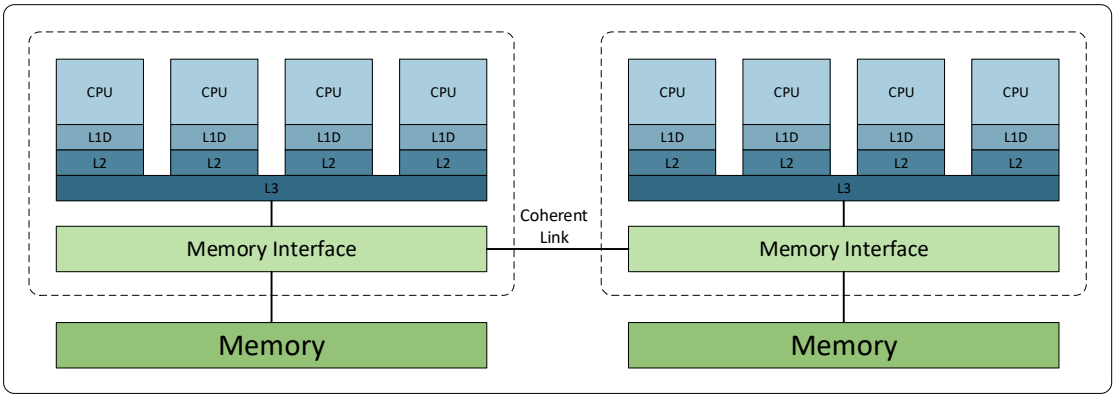

Figure 2.2: Cache-coherent Nonuniform Memory Access architecture.

depicted. All CPUs have access to the same main memory. Figure 2.2 illustrates the more complex cache-coherent Nonuniform Memory Access (ccNUMA) architecture. The example consists of two so-called locality domains connected by a coherent link that allows both processor groups to access the memory of the other domain. The processors of a locality domain share an additional common cache-level (L3). In the context of this work, both introduced shared-memory computing architectures are primarily relevant as building blocks of distributed-memory computers, the second main type of parallel computing that is introduced in the following section.

2.1.2 Distributed-Memory Computing

Distributed-memory computing comes with a more complicated programming model. In this scenario, individual computing nodes only have direct access to their own local memory while using a communication network to facilitate cooperation with other processors. Figure 2.3 illustrates the simplest case of distributed-memory computing, where several processors, each with their own cache and memory, are connected to each other via a communication network [6, p. 102]. The simple exemplary system has four processors, each of which has their own cache and own main memory that none of the

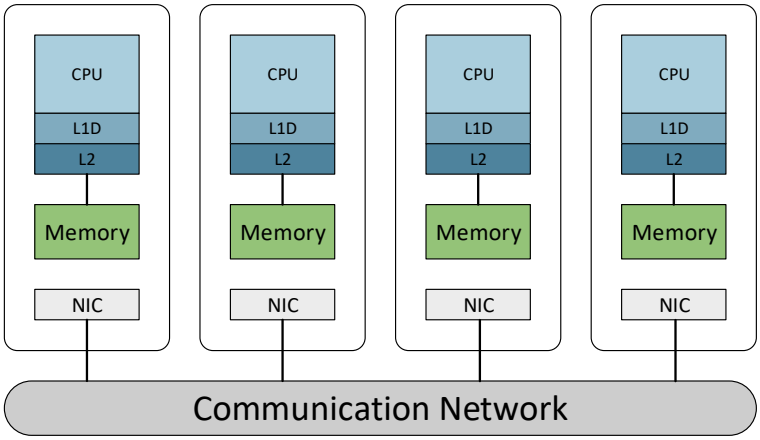

Figure 2.3: Distributed-memory computing architecture.

others can access directly. Processes use network interface cards (NIC) to communicate via a network. This example can give an overview over the general programming model for distributed-memory computers, but has no practical relevance anymore.

Modern, distributed-memory computing resources are structured in a more complex fashion, in particular their nodes are shared-memory computers on their own, making use of several CPUs. The resulting hybrid architecture is shown in Figure 2.4. The depicted simplified programmer's model shows two shared-memory parallel computers connected via a communication network. Modern HPC clusters, which are described in the next section, follow this architecture. Therefore, application developers, who want to make use of all CPU cores present on a HPC cluster, have to consider the distribution of data over the memories. The ability of a program to increase its performance by adding more nodes to a cluster is also referred to as *horizontal scalability*. *Vertical scaling*, in contrast, describes the performance increase achievable by adding more resources to a single existing machine. Vertical scaling is generally easier, as it is only concerned with shared-memory architectures, in which the operating system takes care of accessing the correct memory. The standard for facilitating the necessary communication process between nodes of a distributed-memory computer is the Message Passing Interface (MPI) [7]. However, as will be detailed in later sections of this thesis, we apply an alternative technique to distribute computational workload across multiple cluster nodes and to achieve horizontal scalability.

2.2 High Performance, Cluster and Grid Computing

The goal of HPC is mainly to solve large problems with massive computational complexity by using powerful computing resources. In recent years, HPC has found many use cases also in data-intensive applications [8]. The fields of grid and cluster computing are subsets of the HPC domain. Modern HPC environments can generally be seen as clusters or

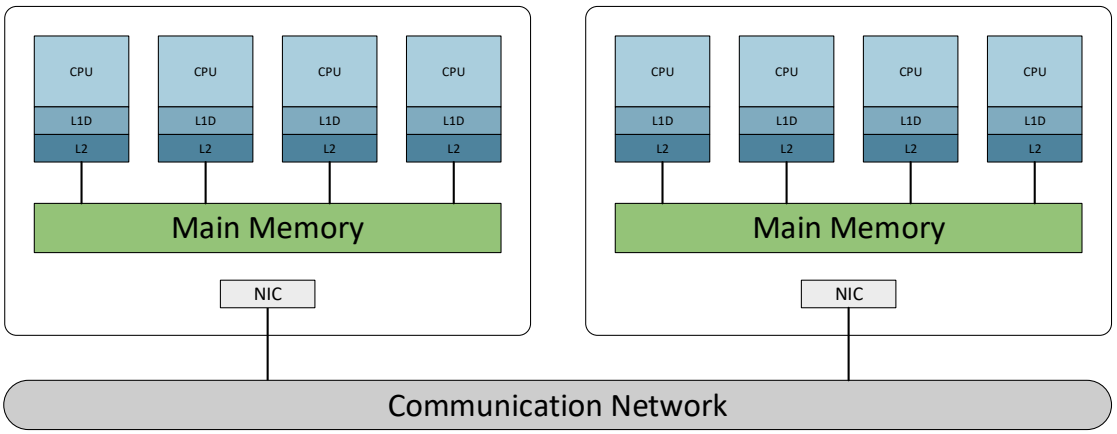

Figure 2.4: Hybrid distributed-memory system.

grids. Both terms imply the usage of connected, but otherwise independent computing nodes. The following paragraph explains the differences before Section 2.2.2 addresses distributed resource management systems (DRMSs), a type of software that is needed to schedule and run jobs on HPC clusters.

2.2.1 Basic Concepts

The terms *cluster* and *grid* are often used interchangeably when it comes to HPC. This is unfortunate, as the original meaning is quite different. A cluster is a collection of computers connected by a high-speed local communication network, such as 10/100-Gigabit Ethernet, Myrinet, or Infiniband [9]. The result is a form of distributed-memory computing following the hybrid architecture shown in Figure 2.4. The individual computers of a cluster, commonly called nodes, work together in order to solve a single problem. This problem or purpose can vary, generally three types of clusters are defined [9]:

- **Fail-over clusters:** Such a cluster uses redundant nodes to deal with system failures. Even if multiple nodes crash, the remaining nodes continue to work as intended, hiding the failure from outside clients.
- **Load-balancing clusters**: In order to provide better overall performance, cluster computing can be used to evenly distribute computational workload among several nodes. A typical use case is a web server cluster that assigns requests to different nodes in order to decrease the average response time.
- **High performance clusters:** This type of cluster, which is of course the most relevant for this work, is used to perform extensive computational operations rather than small IO-oriented tasks, such as answering web service requests. HPC clusters are heavily dependent on a fast communication network connecting the compute nodes. Using high-speed network technologies, such as 100 Gigabit Ethernet or InfiniBand, massively parallel processing becomes possible. This is further supported

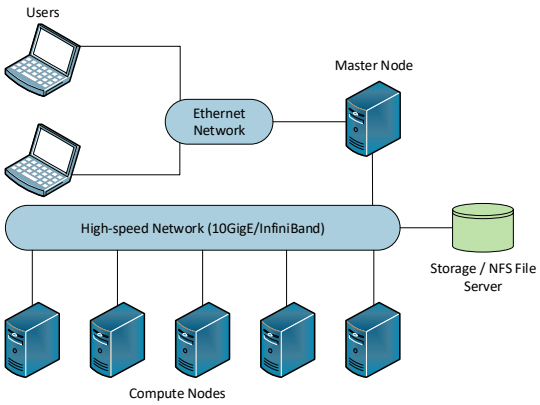

Figure 2.5: Basic HPC cluster setup in which the users do not have direct access to the compute nodes, but have to login using the master node.

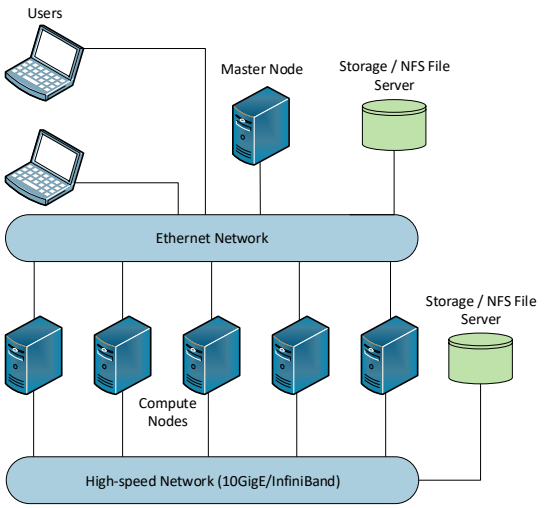

Figure 2.6: Alternative HPC cluster setup in which the nodes are interconnected via a high-speed network but also connected to the regular organization network.

by specialized programming interfaces such as MPI and OpenMP. Furthermore, the term *supercomputer* generally refers to HPC clusters.

Computing clusters usually combine the three above use cases. Consequently, the software platforms described in Section 2.2.2 are able to support all three purposes. Two basic examples for a physical cluster setup are given in Figures 2.5 and 2.6. The former illustrates the basic setup that is present in many organizations. It provides a clean separation between the cluster's internal network and the general organization network. This comes with a downside, as users have to use a gateway node for all operations they wish to perform on the cluster. The second variant compromises on the separation of the main organization network and the cluster network in order to achieve usability benefits. This allows to provide a file storage server to which both the compute nodes and the users have direct access, resulting in easier input and output handling.

Grid computing, on the other hand, is not restricted to fast connected nodes. It only implies the usage of a set of computers that are connected by a communication network in order to reach a common goal [10]. These computers can be loosely connected and stretched over multiple administrative domains such as different companies or organizations [9]. Grid computing software therefore has to be able to facilitate the sharing, selection and aggregation of geographically distributed resources [11]. A grid is often heterogeneous and can incorporate clusters among other types of computing resources [9].

In the context of this work, the focus lies on cluster computing as it is illustrated in Figures 2.4, 2.5, and 2.6.

2.2.2 Distributed Resource Management Systems (DRMSs)

Even though HPC clusters provide large amounts of computing resources, their capacity is still limited. As a consequence, software systems are needed to perform the complex workload management. This includes numerous administrative features such as user management and access control as well as job scheduling and prioritization mechanics [12]. Generally, the functionalities provided by the competing platforms can be grouped into the following categories:

- **Resource Allocation and Job Scheduling:**
  One of the most important tasks of any cluster management and job scheduling system is to govern over the physical resources of the underlying cluster. The software decides, based on configurable parameters and policies further described below, which computational jobs are allowed to run at which time. This introduces the need for user management capabilities. Within an organization, different users may be allocated unequal amounts of resources.
- **Interfaces for Job Execution, Monitoring and Management:**
  Naturally, users must be provided with interfaces to be able to interact with the platform. The most essential tasks are job submission, monitoring of existing jobs, and managing the cluster. The latter includes configuration of compute nodes, adjusting parameters for resource allocation and job scheduling policies as well as altering user rights.
- **Queue Management and Load Balancing:**
  By extending the concept of resource allocation, load balancing and queue management techniques try to equally distribute load among compute nodes, making sure that all nodes are working at their ideal capacity. Furthermore, queue management is responsible for reducing contention, for instance by prioritizing jobs that are expected to run only a short time.

In the following, we will use the term DRMS in order to refer to cluster resource management and job scheduling software. While in theory, different job schedulers may be used with specific resource managers, most systems already come with an integrated job scheduler that is then used by the vast majority of users. For this reason, we define the DRMS term to denote systems that deal with all three of the above defined categories. Figures 2.7 provides an abstract representation of such a system.

The most prominent DRMSs are Torque/PBS [13, 14], OpenLava [15], IBM Platform LSF [16], HTCondor [17], Slurm [18], Univa Grid Engine [19], and several open source derivatives of the former Sun Grid Engine (SGE) [20]. It is not rewarding to review these products individually, instead, the following parts of this section introduce the

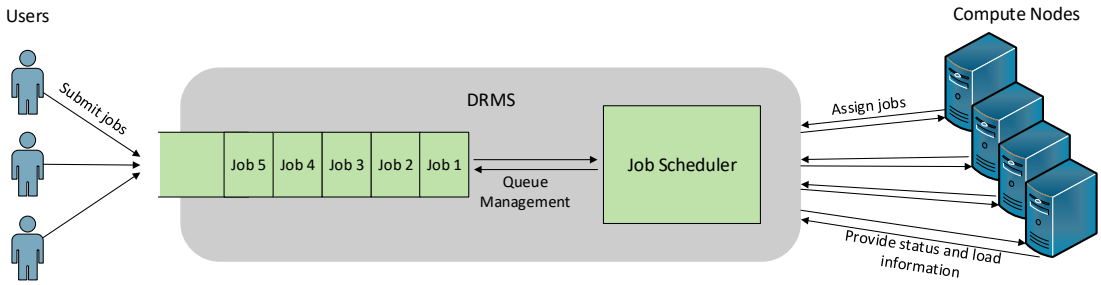

Figure 2.7: Job scheduling on distributed-memory clusters.

most important approaches used in this field. Furthermore, the specifics of the Sun Grid Engine (SGE) and its modern derivatives are discussed. SGE is the DRMS that is used in the practical part of this thesis, and some of its principles are therefore important in order to comprehend our design decisions.

Scheduling and Management Concepts

Scheduling approaches can generally be categorized in space-sharing and time-sharing policies. Further advancements are variants of these and often combine elements of both [21]. A space-sharing policy assigns resources to a job until it is completed. The most prominent examples for space-sharing algorithms are First Come, First Served (FCFS), Round Robin (RR), Shortest Job First (SJF), and Longest Job First (LJF). These are universal concepts in computer science and should not require further introduction. The disadvantage of a basic space-sharing policy is, that only the current state of the queue is considered and resources may remain idle, even though a later submitted job could utilize them. Figure 2.8 illustrates this problem. The depicted FIFO-based space-sharing queue is unable to fill idle slots with small, lower priority jobs, because it waits until enough resources are available to start the high priority jobs.

Time-sharing policies do not assign resources for the entire duration of a job, but rather distribute discrete time intervals, in which a resource can be used, to competing jobs. This increases fairness and solves the issue portrayed in Figure 2.8, where long, resource-intensive, high-priority jobs block the execution of short jobs that could actually be executed earlier without delaying the others. However, there is one major drawback, the execution of jobs that have outrun their currently assigned time intervals has to be paused, which causes a massive overhead. Therefore, time-sharing policies are not used by most modern job schedulers. Figure 2.9 shows a time-sharing queue that can can easily fill up idle CPU time with later submitted jobs. This does not affect jobs with higher priority, because the rearranged jobs will only occupy the specified time interval. Therefore, compared to the simple space-sharing policy depicted in Figure 2.8, there is less CPU idle time. Nevertheless, the total amount of time needed before all jobs are finished is longer, since the overhead introduced for pausing and resuming jobs lengthens

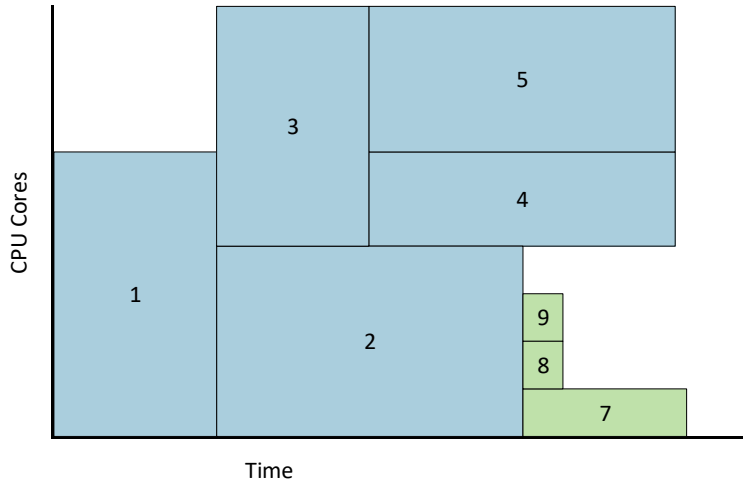

Figure 2.8: FIFO-based space-sharing queue. The numbers indicate the job submission order, which, in this case, also implies job priority. Jobs 7, 8, and 9 could be executed alongside job 1, but are not because they were submitted last and therefore have the smallest priority. This may cause wasted resources.

their execution times. Also, slots have to be used completely, meaning that a job that finished at the beginning of an assigned time interval causes additional idle time.

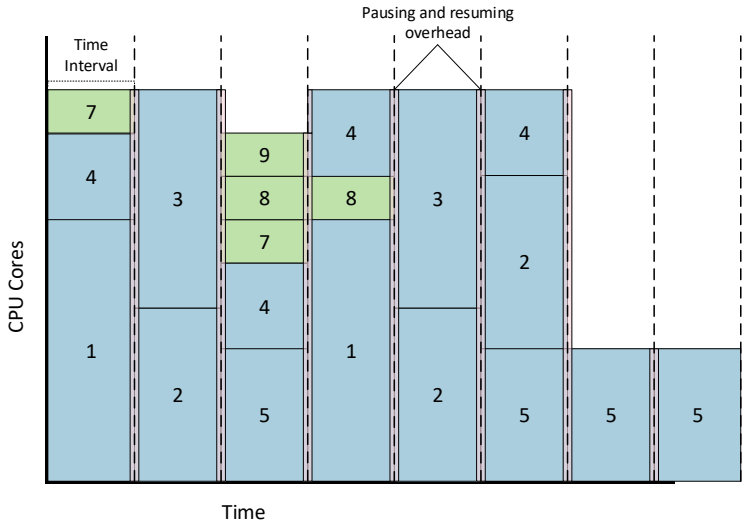

Figure 2.9: Time-sharing queue. Once again, the numbers indicate the job submission order and, in this illustration, also help to identify in which time slots a particular slot is allowed to run.

The most established approach for coping with above mentioned problems is to use space-sharing policies with backfilling [22, 23]. The latter technique allows a job A, which

only needs relatively few resources, to jump in front of a more demanding job B with a higher priority, provided that it will not delay the start of job B. Figure 2.10 shows this process on the basis of the already introduced queue of Figure 2.8. Naturally, this requires the job scheduler to be aware of the expected duration of a job. An appropriate estimation has to be specified by the user, which complicates the job submission process.

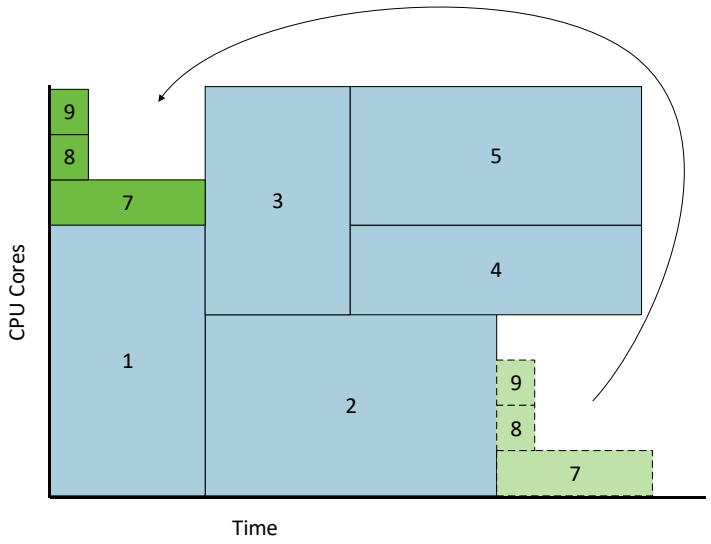

Figure 2.10: Backfilling allows jobs 6, 7, and 8 to jump ahead because they are expected to finish already before jobs 2 or 3 can start and therefore will not delay these higher prioritized tasks. However, they might delay the start of higher priority jobs if job 1 finishes early.

The given introduction only touches the most important techniques carried out by a DRMS. Many specializations and extensions exist, which can not be described here in full detail and therefore we refer the reader to an extensive body of literature [24, 25]. In the context of this work, the job scheduling policies and options of the common resource managers are important, because when the goal is to integrate the compute capabilities of HPC clusters into GUI applications they have to be taken into account and it is necessary to provide mechanics for setting appropriate parameter values. Section 4.2 further explains these practical implications.

Sun Grid Engine (SGE) and its Derivatives

The Sun Grid Engine was initially developed by Gridware, which was later acquired by Sun Microsystems, hence the name. With the acquisition of Sun Microsystems by Oracle, the name was once again changed, this time to Oracle Grid Engine. In 2013, Univa acquired the intellectual property for the technology and continues to market it as the proprietary Univa Grid Engine. However, the project was open source until 2010, when

Oracle acquired Sun and commercialized the code. As a consequence, several open source forks exist to this day. Due to the fact that these offer essentially the same features and most importantly expose the same command-line interface, they are still collectively denoted as SGE. The most important examples of these derivatives are the Son of Grid Engine [26] and the Open Grid Scheduler [27]. It is difficult to assess the market share of SGE and its derivatives, as we have found no respective studies. Nevertheless, it is clear that SGE is still commonly used, especially in the life sciences field. This is illustrated by many projects in the field that focus on supporting SGE-based environments, such as the software package by Kauppi et al. [28]. Furthermore, the Department of Pharmaceutical Chemistry at the University of Vienna supported the creation of this thesis by providing access to an on-site HPC cluster for testing purposes. This system is described further in Chapter 6 and also relies on SGE. Additionally, SGE is available in the official Debian and Ubuntu software repositories and is the default system used by many toolkits that facilitate HPC cluster creation in the cloud, most importantly CfnCluster [29], which will be discussed in detail in Section 2.3.4. Please refer to Section 2.3.3 for more details on cluster computing in the cloud, which is of significant importance in the context of this work.

SGE differentiates between so-called *master* and *compute* nodes. A regular SGE cluster consists of exactly one master node and an arbitrary amount of compute nodes. The former monitors resource usage and performs job scheduling operations. The compute nodes carry out the actual computational work [30]. The master node can also act as a compute node, but usually this is not the case. Instead, its remaining resources are often used to run other services and applications that facilitate HPC cluster operation. This can include various monitoring software and also the server application developed in the course of this work. Job submission can be done on any submission host, which usually means both the master and the compute nodes.

Similar to other resource managers and job schedulers, SGE provides a command-line interface for job submission, monitoring, controlling, accounting and management purposes. Additionally, there are SGE bindings for the *Distributed Resource Management Application API* (DRMAA) [31] for multiple programming languages. This enables direct programmatic interaction with a SGE installation, a feature that is used for the implementation part of this work.

2.3 Moving High Performance Computing (HPC) to the Cloud

Before going into detail about current possibilities of HPC in the cloud, it is important to define some general important concepts and terminology. This is done in Section 2.3.1. Then, in Section 2.3.2, specific cloud offerings for the HPC domain are elaborated, most importantly Amazon's CfnCluster, which is also used for the practical part of this thesis. Finally, Section 2.3.2 discusses performance differences between HPC in the cloud and traditional on-site clusters.

2.3.1 Cloud Computing Overview

The tremendous success of cloud computing in recent years has not been able to eliminate the general confusion about what exactly qualifies as a cloud. In general, the term refers to applications and infrastructure offered as a service over the internet [32]. These variants are typically classified as either Software as a Service (SaaS) or Infrastructure as a Service (IaaS).

Both types offer significant benefits over more traditional forms of computing. A central aspect is elasticity, meaning that cloud computing enables users to add or remove resources quickly and at a fine grain [32]. As a result, the consumed resources can be matched precisely to the current workload. This leads to massive energy and cost savings compared to buying physical hardware. In order to cope with the highest possible load, on-site data centers are usually over-provisioned and, on average, server utilization in these private facilities only reaches a range from 5% to 20% [33, 34]. In contrast, cloud providers offer pay-as-you-go pricing models that imply fine-grained charging based on actual use [32].

In order to provide the mentioned elasticity advantage, cloud computing data centers have to be very large and rely heavily on virtualization technology. Data centers that are only used within an organization or business are not considered to be cloud infrastructure unless they meet these criteria [32].

2.3.2 Service-oriented Offerings

Harnessing the unique advantages presented in the previous section, cloud providers have largely overtaken fields such as data storage, web hosting, and classical SaaS applications as for instance Email. However, the case for HPC in the cloud has been and continues to be more difficult. To date, a majority of scientific HPC tasks are still done using private on-site clusters [1]. The reasons for this are manifold, the most important one is communication network performance, a topic discussed in detail in the next section. However, another reason is that cloud providers have been relatively slow to develop extensive HPC offerings for the scientific community. This has changed in most recent years, and there are now various possibilities for exploiting the HPC capabilities of public clouds in order to perform scientific experiments. Most of them do not give the user direct access to an HPC cluster, but rather take a more service-oriented approach. These options are covered in the following paragraphs.

MapReduce and Hadoop

The MapReduce programming model, introduced by Google in 2004 [35], has quickly found its way into the standard repertoire of distributed computing techniques. It enables easy processing of large data sets on massively parallel systems. Application developers only have to specify a *map* and a *reduce* function, while the underlying implementation takes care of partitioning the data, scheduling execution across a distributed set of machines, and other distributed computing tasks. The map operation processes key/value pairs to generate new intermediate key/value pairs applying developer-defined algorithms. The

reduce function merges all intermediate results from the map operation with the same key. Dean et al. [35] show that many real world problems are expressible in this model.

Given the benefits of the technique, it is very well suited for cloud computing. As a consequence, all major providers offer specialized products for performing large MapReduce tasks as part of their HPC product range. Most often, large-scale MapReduce is done on Hadoop clusters [36], which scale well to arbitrary problem sizes and are designed to support the MapReduce programming model. Compared to other distributed computing frameworks, Hadoop is particularly focused on massively parallel big data applications without extensive communication overhead. For this purpose, a specialized file system was developed, namely the Hadoop Distributed File System (HDFS) [37], which excels at storing extremely large amounts of data.

The most prominent product offerings related to MapReduce and Hadoop are Amazon Web Services Elastic MapReduce (EMR) [38], Microsoft Azure HDInsight [39], and Google Cloud Platform Cloud Dataproc [40].

Batch Scheduling as a Service

HPC resources are still rarely offered at an infrastructure level. Instead, cloud providers heavily promote batch scheduling as a service. The benefits of these services over directly managing a cluster with a corresponding resource manager are obvious. In particular, the service provider typically manages the configuration, handles the operating system level, and provides web-based interfaces for monitoring and customization demands. These features overlap with the concepts introduced in Chapter 3 and can help provide access to HPC resources for a mainstream audience.

However, the approach of batch scheduling as a service also comes with significant limitations. Users are greatly limited in regards to the type of jobs they can run. For instance, the Batch product marketed by Amazon Web Services [41] requires customers to define their jobs via docker containers using special configuration files. In these containerized environments, distributed-memory programming tools such as OpenMP and MPI are either not supported or vastly more complex to set up than with regular cluster installations [42], although this may change in the future. These downsides pose substantial obstacles for research institutions looking to transition from physical on-site clusters to elastic HPC resources in the cloud. Furthermore, both scientists and application developers in the scientific HPC domain are already used to clusters running conventional scheduling software as described in Section 2.2.2. New interfaces, even though simpler in theory, are hard to integrate with existing workflows and are therefore often not desirable for this audience. Alternatives that take this into account are discussed in the following segments.

2.3.3 Cluster Computing in the Cloud

In recent years, several companies have started to provide tools for building general purpose HPC clusters in the cloud. These should, in theory, combine the benefits of

cloud computing with the flexibility and wide-range of possibilities conventional HPC clusters provide (cf. Section 2.2).

In principle, IaaS offerings like Amazon's elastic computing cloud (EC2) have always been available for building HPC clusters. This is because IaaS, per definition, allows the customer to take full control over the used operating systems and application software. By exercising this power, customers can simply install cluster and grid management software on their rented infrastructure and turn the virtualized machines into cluster compute nodes. The problems with this approach are the required manual configuration and setup work, coming close to the work needed for installing an on-site cluster, and even more the inability of cloud computing providers to guarantee a fast interconnection network between compute nodes. The former has since been overcome. Several companies and organizations have developed tools to facilitate the automatic setup of HPC clusters in the cloud. These include the Massachusetts Institute of Technology (MIT), which has developed StarCluster [43], a toolkit for automating the process of building, configuring, and managing clusters of virtual machines on the Amazon Elastic Compute Cloud (EC2) cloud. Furthermore, Univa offers software and support for running their Univa Grid Engine using IaaS products of the largest cloud providers, including Amazon, Microsoft, Google and VMware. Cycle Computing has a similar approach, also supporting all of the most important cloud providers but relying on free third-party DRMSs (SGE, OGE, Condor, Torque). A freely available toolkit that supports multiple cloud providers is ElastiCluster [44, 45]. Similar to StarCluster, it originates from a research institution, namely the University of Zurich, but offers a more sophisticated feature palette, ranging from Hadoop support to more extensive automated setup options.

However, the remaining problem of slow interconnection networks between compute nodes remains. For instance, Amazon does not provide specialized high-speed cluster interconnection networks such as InfiniBand. The current high-end offer is a 20 Gigabit Ethernet network. While this is already tolerable for HPC, in reality, Amazon and other providers often fail to guarantee these figures [46, 47]. This is not surprising as virtualization and other essential cloud technologies impose additional challenges on high-speed networking. Section 2.3.5 discusses the current performance characteristics of HPC systems in the cloud in detail.

2.3.4 Amazon CfnCluster

CfnCluster (*CloudFormation Cluster*) functions very similar to StarCluster and Elasticluster. It is maintained directly by Amazon Web Services (AWS) and developed as an open source project accessible on GitHub [29]. While the restriction to AWS is certainly disadvantageous, the official support from and for a specific cloud provider results in a stable and regularly updated product. The latter is especially important for any framework facilitating cluster computing in the cloud, because the tool has to be able to deal with new versions of operating systems, updates to cluster computing software and changes in the product palette of the supported cloud providers.

The CfnCluster software is essentially a command-line tool written in Python that provides simple commands for creating, updating, stopping, starting, and deleting HPC clusters in the AWS EC2 cloud. Naturally, to provide these functionalities, the locally installed CLI has to use the web APIs of existing AWS services. The most important one of these services is the eponymous AWS Cloudformation. It facilitates the creation and management of a collection of related AWS resources. A set of nodes belonging to a specific HPC cluster can be such a collection. Another essential service is *Auto Scaling*, providing dynamic elastic scaling of the number of compute nodes depending on the current workload. As a consequence, CfnCluster can be configured to automatically add nodes when jobs are pending and not able to run due to the cluster being fully utilized.

Section 5.4 describes the concrete CfnCluster setup utilized for the deployment of the software developed in the course of this work, while Chapter 6 contains experimental results gathered through the usage of CfnCluster.

2.3.5 Performance Compared to On-Site Clusters

HPC in the cloud faces complex challenges that have to be overcome in order to compete with on-site clusters. Specifically, these include the already mentioned interconnection networks, which are most often not built for supporting HPC applications, and secondly, a performance decrease due to virtualization.

Already in 2009, Napper et al. [48] have used the LINPACK benchmark [49] to assess the performance of HPC clusters allocated in the Amazon EC2 cloud. The sobering conclusion has been that, at least for communication intensive problems, HPC in the cloud can not yet make use of its potential strengths regarding scalability and cost saving. To the contrary, it has been shown that the achieved floating point operations per second (FLOPS) *per dollar spent* decrease exponentially as the allocated cluster grows. This is, as expected, due to the slow interconnection network and limited memory available on single nodes.

One year later, in 2010, Jackson et al. [50] have performed similar experiments and also encountered the familiar problem of slow network connections. They observed a clear correlation between the runtime of different distributed applications and the amount of time spent communicating. However, Jackson et al. [50] have also shown that for applications requiring little communication between computing nodes, HPC cloud clusters can scale well.

The situation has since progressed further. Real world problems are often embarrassingly parallel or come close to it, enabling all benefits of cloud computing, even with sub-par interconnection networks. This development is illustrated by various case studies of different scientific fields, especially the computational life sciences. Novartis used the Amazon EC2 cloud with 87 000 on-demand CPU cores to vastly decrease computation time for a VS experiment [51], a scientific technique explained in detail in Section 2.5. HGST built a similarly strong cloud cluster of 70 000 cores to run a simulation for finding an optimal hard drive head design [52].

It is clear that HPC in the cloud has become a reality. While virtualization and limited communication network speeds are still problems, providers are likely to introduce more and more specialized offerings dealing with these obstacles. Many embarrassingly parallel real-world problems, apart from benchmarks such as LINPACK, are already well suited for being executed on cloud-allocated clusters. VS of compound libraries is an ideal example for problems with minor communication demand between distributed-memory nodes and is therefore well-suited to be run on cloud clusters.

2.4 LigandScout

In the context of this work, the LigandScout graphical desktop application serves as a concrete example for applications that aim to support scientific research and that can profit from built-in, seamless access to remote HPC resources. In the practical part, the focus lies on providing this transparent integration for LigandScout and its VS implementation, hence, it is necessary to cover the main concepts behind the application and introduce its most important use cases.

The software was initially released in 2005 and since then has matured to become a feature-rich molecular modeling and design software suite. The initial goal was to provide the first fully automated method to create a pharmacophore model from a known 3D protein structure [53]. The International Union of Pure and Applied Chemistry (IUPAC) defines a pharmacophore as "an ensemble of steric and electronic features that is necessary to ensure the optimal supramolecular interactions with a specific biological target and to trigger (or block) its biological response" [54]. Figure 2.11 depicts a pharmacophore within the corresponding binding pocket. Both the image and the pharmacophore have been created using the LigandScout software.

LigandScout aims to reduce costs in the early stages of the drug discovery process by enabling in-silico experiments that are vastly cheaper than their *in vitro* counterparts. The latter require manual synthesis of chemical compounds or acquisition from external providers. The most prominent example for cost reduction in drug discovery research is VS, a process that fits the pharmacophore concept very well.

The LigandScout software has been released and is now maintained by Inte:Ligand GmbH, a software company based in Vienna. This work is conducted in collaboration with Inte:Ligand, which provides access to the source code of LigandScout and expertise regarding software development for the life sciences community.

2.5 Virtual Screening (VS)

VS is a prominent in-silico technique to aid the drug discovery and design process. In the first part of this section, a general overview of the technique is given before specifics of the screening process in LigandScout are presented. Then, the problem is formally described, providing definitions that will also be used in later parts of this work. The chapter closes

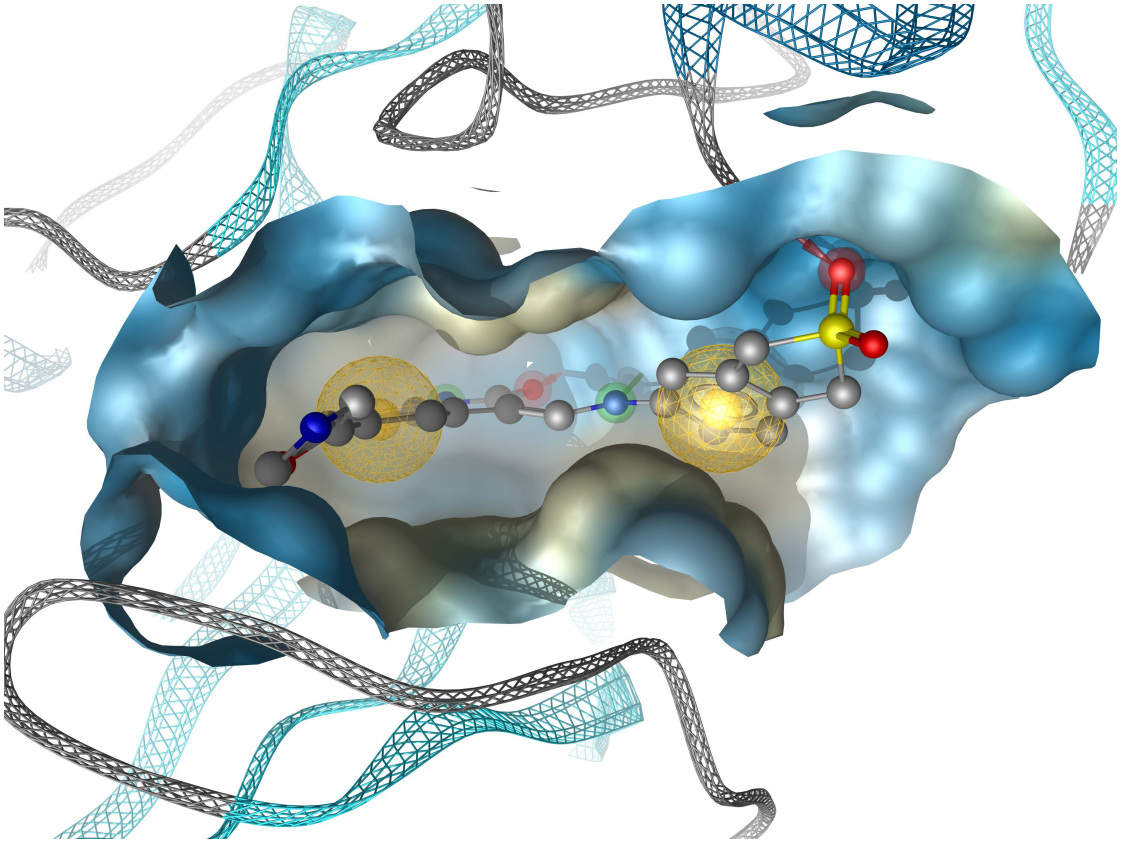

Figure 2.11: 3D Rendering of a LigandScout pharmacophore model, where the pharmacophore is shown with its corresponding ligand and is embedded in the respective protein binding site.

with an explanation of how the general VS process as well as the specific algorithm used in LigandScout can be modified to allow a parallel and distributed execution.

2.5.1 Overview

The Screening of chemical libraries to obtain compounds that show activity towards a specific biological target has been a standard process in pharmaceutical research for several decades. The traditional approach of empirical screening relies on expensive *in vitro* experiments, meaning that the molecule libraries have to be bought or synthesized before being tested for their biological activity [55].

VS on the other hand, tries to massively decrease the cost of screening by using computer-based methods. Already introduced in the 1970s, it has since matured to become a widely used routine to support the drug discovery process. While it is still not able to compete with experimental screening regarding the false-negative and false-positive rates, VS is able to deal with far larger compound database sizes [55].

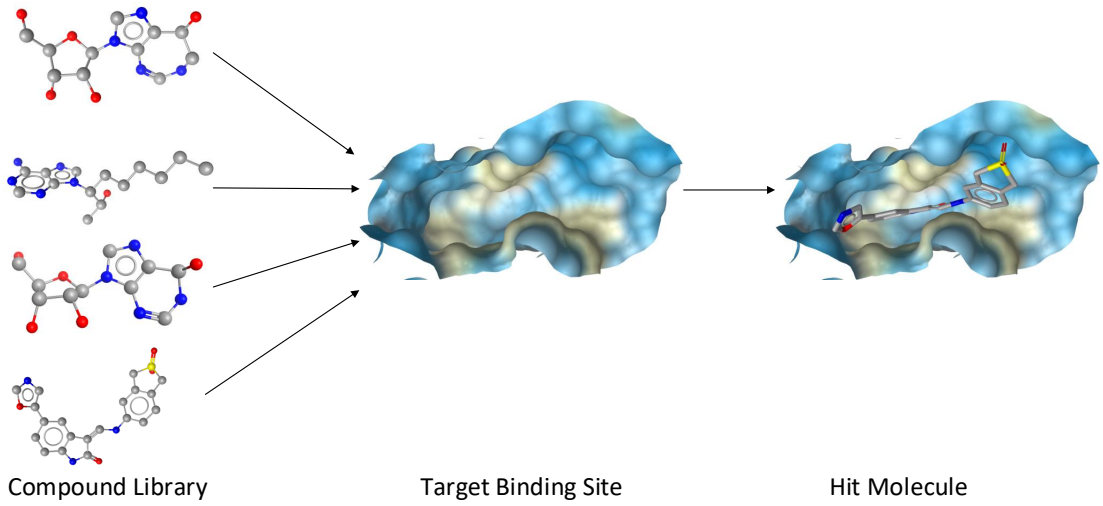

Figure 2.12: General concept of the VS process.

The process of using computational resources to determine whether a particular molecule is a possible ligand for a specific target is a research field on its own, located at the intersection between computer science, chemistry, pharmacy and biology. The different approaches can be categorized into two types, docking and pharmacophore-based VS. The first one aims to *dock* molecules of the compound libraries into the original target binding site. This is done by applying a scoring function measuring the likelihood of the compound binding with the target. In general, many iterations are needed for testing one particular molecule, each time slightly adjusting the molecule position in the binding pocket [56]. Figure 2.12 illustrates the general task of checking the compounds of a potentially large library against a target binding site and how a matching molecule fits the target. Successfully matched molecules are commonly called hits or hit molecules. The pharmacophore-based VS, on the other hand, relies on analyzing the target binding site before the actual screening process and on creating a pharmacophore model in the process. During the actual screening process, the library compounds are then only checked against the pharmacophore model. This results in lower execution times, as the target binding site only has to be analyzed once, but the results are, of course, heavily dependent on the model creation process. The described workflow is portrayed in Figure 2.13. It is also shown how a hit molecule fits to a specific pharmacophore. This work relies on the pharmacophore-based VS algorithm used in LigandScout, which is explained in more detail in the below section.

In case more than one target is used in parallel, the screening process is often called *activity profiling*. This is another generic drug design technique, that can be specialized to fit many purposes. Schuster [57] gives an excellent overview as to how pharmacophore models can be used in this field. Generally, a pharmacophore model collection is set up, representing both wanted and unwanted interactions. Selected molecules or even a database of compounds can then be screened against this compilation of models. Ideally, a

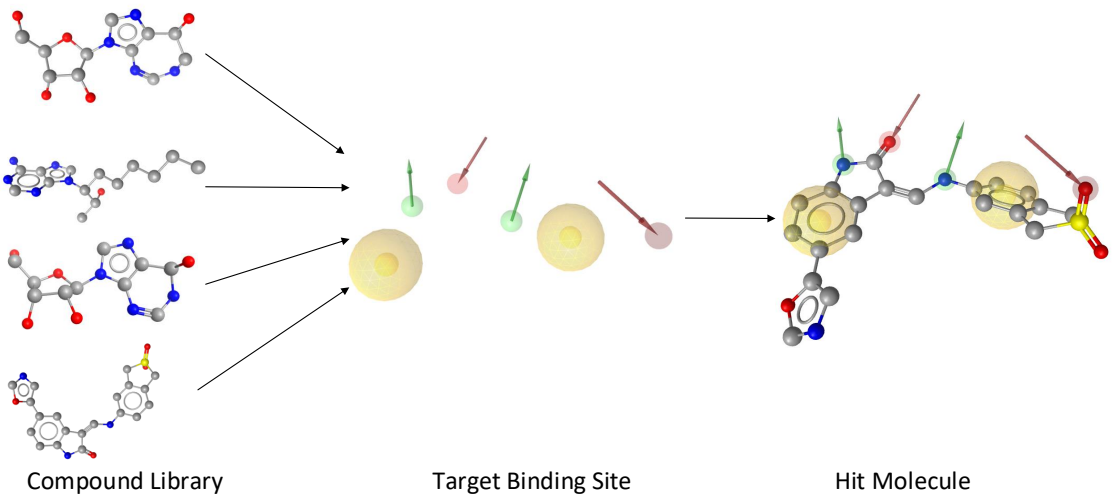

Figure 2.13: Virtual screening by use of pharmacophores. The pharmacophore model represents the target binding site. The compounds of the library only have to be matched against the model instead of the actual binding pocket.

comprehensive profile of a ligand's biochemical activity is generated. Naturally, choosing the correct pharmacophores for the task at hand is crucial. Docking based variants are less prominent in activity profiling scenarios, because the greater computational complexity can make the parallel usage of many targets infeasible.

2.5.2 Virtual Screening in LigandScout

Even though LigandScout also has built-in docking capabilities, it has a strong focus on pharmacophore-based methods. The workflow generally starts with creating the query 3D pharmacophore model, a process not described further in this work. The actual database screening is then implemented as a multistep filtering process. Pre-filtering takes place by eliminating compounds that do not match basic requirements such as the query pharmacophore's feature-types and feature-counts or do not stand quick distance checks [58].

After pre-filtering, 3D-matching algorithms are used. Generally, these algorithms are much slower but more restrictive. The final decision whether a compound is a potential ligand or not is eventually made in this step, therefore the quality of the screening results directly depends on the classification algorithms used here. To make an informed decision, a geometric 3D alignment of the pharmacophore and the molecules to be screened is necessary. One approach that is used for this purpose is minimizing the root-mean-square deviation (RMSD) between corresponding feature pairs [58]. The minimum value is then compared to a threshold in order to decide whether the screened molecule matches the pharmacophore.

The generation of different conformations for each molecule is not done during VS, instead LigandScout expects input compound databases that already cover the conformational space of each contained molecule. The creation of such databases from a set of compounds is therefore called conformer generation. This process comes with extensive computational demands on its own but has to be performed only once. The resulting databases can then be used for an arbitrary amount of screening experiments.

Apart from early stage drug discovery, VS also plays an important role when it comes to validating and refining pharmacophore models. By using compounds of which the biochemical activity is already known, several validation measures can be computed [58] and used to assess the query model. Sensitivity (Se) is one possibility and gives the ratio of the number of true positive compounds TP to all active molecules in the screening database, which are equal to the sum of true positives and false negatives FN. It is defined as

$$Se = \frac{TP}{TP + FN} \quad . \tag{2.1}$$

Specificity (Sp) represents the counterpart and is the ratio of the number of correctly ruled out compounds (TN) to all inactive compounds, which are consequently given by the sum of true negatives TN and false positives FP. Therefore, it is defined as

$$Sp = \frac{TN}{TN + FP} \quad . \tag{2.2}$$

Another and more sophisticated technique for measuring a model's validity are Receiver Operating Characteristics (ROC) curves [59]. This graphical method displays the increase of false positives that comes with extracting more true positives by lowering the threshold parameters for classifying a molecule as active. For this purpose, the false positive rate is shown on the x-axis and plotted against the true positive rate on the y-axis. The line at 45 degrees represents a random classifier that any meaningful approach should be able to outperform. The area under the curve (AUC) can be used as an additional metric, a larger area indicates a better screening performance. ROC curves are one of the main tools offered by LigandScout to assess the success of a VS run. Figure 2.14 depicts a screenshot of the LigandScout screening perspective along with a resulting ROC curve plot. As can be seen, the area under the curve is large, indicating that the used pharmacophore model is well designed.

2.5.3 Formal Description

In order to provide a universal taxonomy that can be used in the remaining parts of this thesis, a formal description of the VS process is given.

A screening job can use an arbitrary amount of pharmacophore models, although using multiple models is, as already stated, often called *activity profiling* instead of screening. The pharmacophore models are given by $\{m_1, m_2, ..., m_o\}$, where o denotes the number of models. Furthermore, an arbitrary amount of n input compound database files can be provided as an input, and this set of files is denoted as $F = \{f_1, f_2, ..., f_n\}$. A specific

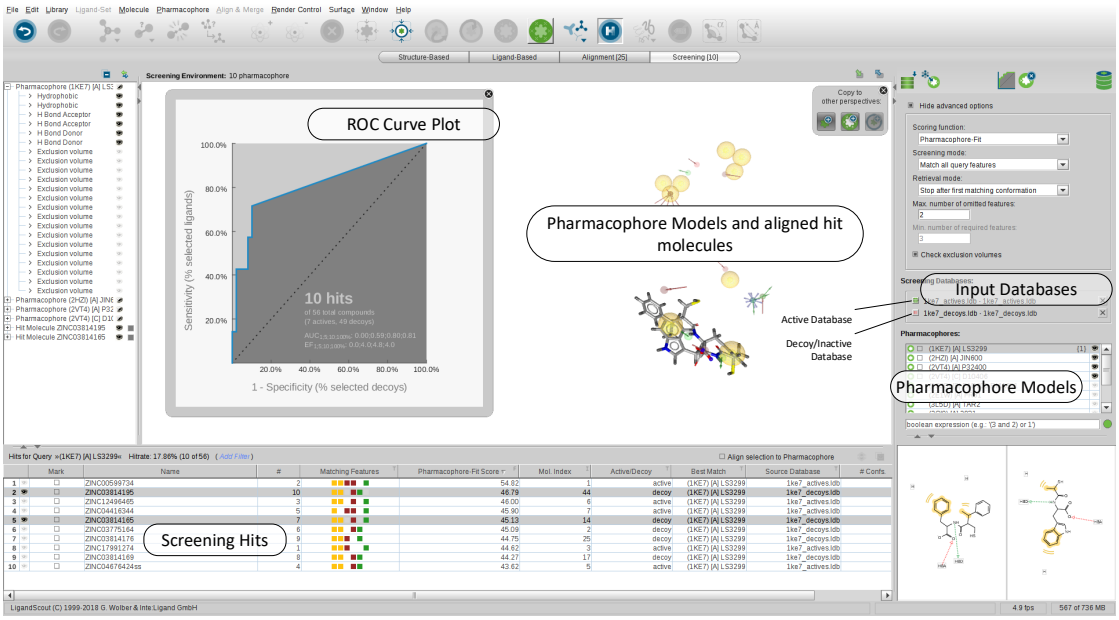

Figure 2.14: LigandScout screening perspective with results of a successfully finished virtual screening experiment.

file f_i contains c_i compounds, where $i \leq n$. As a consequence, the total amount of compounds in all databases is

$$C = \sum_{i=1}^{n} c_i \quad . \tag{2.3}$$

The set of all compounds is defined as $D = \{d_1, d_2, ..., d_C\}$. We can now define a virtual screening task s as a function, which takes as inputs a pharmacophore model m_i and a compound d_j, and it determines whether the given compound matches the input model, i.e.,

$$s(m_i, d_j) = \begin{cases} 1, & d_j \text{ is a hit against model } m_i; \\ 0, & d_j \text{ is not a hit i.e. it does not fit model } m_i \end{cases} \quad . \tag{2.4}$$

In case there is only one query pharmacophore model to screen against, the total number of tasks T is equal to the number of compounds C, considering the more general case of an arbitrary number of models o, the number of individual screening tasks is

$$T = o \times C \quad . \tag{2.5}$$

All individual tasks are independent from each other, most importantly, they do not need to share data. Every task only needs one pharmacophore model and one compound to check for a biological activity. The task itself is then comprised of analyzing the activity of the compound in question towards the target represented by the pharmacophore model. This, as explained in previous sections of this chapter, represents a wide research topic on its own. For the sake of this definition, it is only relevant that every task can potentially

report a single hit, meaning that the compound corresponding to the respective task fits well to the query pharmacophore and is a candidate for biological activity against the target represented by the pharmacophore model. The total number of hits H is therefore in any case smaller or equal to T. However, in practice, the number of reported hits H will be much smaller than T. Otherwise the used pharmacophore models were not specific enough.

A complete screening experiment S is defined as the set of all tasks that can be formed with the given input compounds and input models:

$$S = \begin{Bmatrix} s(m_1, d_1) & s(m_1, d_2) & \dots & s(m_1, d_C) \\ s(m_2, d_1) & s(m_2, d_2) & \dots & s(m_2, d_C) \\ \vdots & \vdots & \ddots & \vdots \\ s(m_o, d_1) & s(m_o, d_2) & \dots & s(m_o, d_C) \end{Bmatrix} \quad . \tag{2.6}$$

If a specific compound d_x matches any of the input pharmacophore models, it has to be included in the final results. Consequently, d_x is present in the results iff

$$\sum_{i=1}^{o} s(m_i, d_x) \geq 1 \quad . \tag{2.7}$$

Every hit has metrics assigned that depend on the particular screening algorithm in use. The metrics for tasks that did not yield a hit can be discarded. The total number of hits H can subsequently also be defined as

$$H = \sum_{i=1}^{o} \sum_{j=1}^{C} s(m_i, d_j) \quad . \tag{2.8}$$

2.5.4 Parallel Execution

VS is a classical example of an embarrassingly parallel problem, which means that it can easily be divided into concurrently executable subproblems [60, p. 14]. As defined in Section 2.5.3, the obvious approach for parallelizing VS is to regard the calculations for each compound as independent tasks.

In principle, these tasks can be divided further because interactions between individual atoms of the molecule to be screened and parts of the binding site can often be analyzed separately. Guerrero et al. [61] present approaches for doing this using different methods such as MPI, OpenMP, Hybrid MPI-OpenMP, and CUDA programming models. They show substantial speedups on a per-compound level for docking-based VS. However, in the context of pharmacophore-based VS, these efforts are largely futile, because the calculations for a single molecule are done in much less than one second. Therefore, such approaches are not implemented in LigandScout and not further discussed in this work.

The communication and coordination overhead of pharmacophore-based VS divided on a per-compound level is small. It comprises of two things:

1. Assigning molecules to be screened from the input databases to the individual threads.
2. Gathering of hits from all threads along with scoring metrics.

In principle, these coordination tasks would be well suited for an MPI-based implementation. However, LigandScout's VS command-line application is not an MPI program and therefore does not support distributed-memory computing itself. To circumvent this, the server application developed in the course of this work takes care of splitting large VS jobs into smaller sub-jobs to be executed on individual cluster nodes. This also includes result aggregation and comes with the important benefit that sub-jobs can start to run independently once any cluster node has free resources, a feature that is particularly important in the context this work, in order to provide a graphical user interface with direct feedback. An in-depth discussion of the implemented approach is given in Sections 4.3 and 5.2.

CHAPTER 3

Related Work and State of the Art

As already outlined, working with remote HPC clusters can be cumbersome for various reasons. The most apparent obstacle is the requirement to submit computational jobs or workflows via the command line, a process that is unfamiliar to many scientists without formal computer science education. Even if the requirement of command-line expertise is no obstacle, submitting jobs is tedious. Input files have to be exported from GUI applications, manually transferred to the file storage of the computing cluster, and a potentially large amount of output files has to be managed. Figure 3.1 illustrates the resulting workflow. Parts of the problem can be mitigated by cluster setups that provide users with direct access to the file storage of the HPC cluster in question from their workstation computers. Such a system is illustrated in Figure 2.6. In particular, this topology circumvents the need for the tedious manual transfer of files over a network. However, the challenge of managing a potentially large amount of output files remains, as does the need for importing and exporting files to and from GUI applications that are used to manipulate the data.

Conventional scripting approaches can only provide limited improvements. Exporting data from GUI applications can usually not be automated due to a lack of external APIs, the same is true for importing back result files. Furthermore, requiring programming knowledge decreases the accessibility of HPC resources for researchers without a background in computer science. Therefore, different solutions are required. This chapter presents existing approaches to simplify working with HPC clusters and summarizes open research questions.

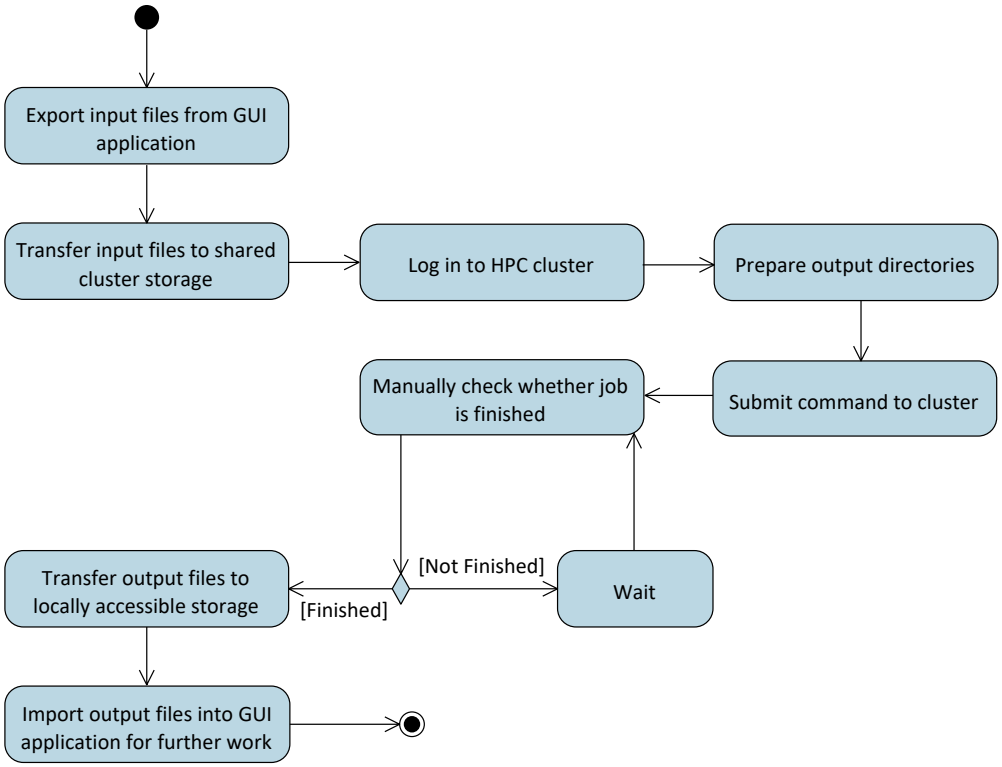

Figure 3.1: Basic HPC cluster computing workflow. The activity diagram shows the typical steps necessary to perform a computational job on a remote HPC cluster. Input and output files are handled manually.

3.1 Simple Front-Ends to Distributed Resource Management Systems (DRMSs)

There are various tools that provide web-based front-ends for DRMSs such as SGE. The simplest variants offer only monitoring capabilities without support for submitting or canceling computational jobs.

Software systems such as CHReME [3] and the Moab HPC suite [62] are exemplary tools that go beyond providing accounting information. Through their web-based interfaces, it is possible to submit jobs and to manage them while they are running. Figure 3.2 depicts the job submission workflow that results from these features. There are many examples of similar software systems, however, most of them have in common with CHReME and the Moab HPC suite that they are commercial and financially out of reach for many scientific institutions.

An exception to this circumstance is Yabi [63], which also offers a feature-rich web-interface for most resource manager systems and which is fully open source. Many of its GUI elements are drag-and-drop based, enabling users from different backgrounds

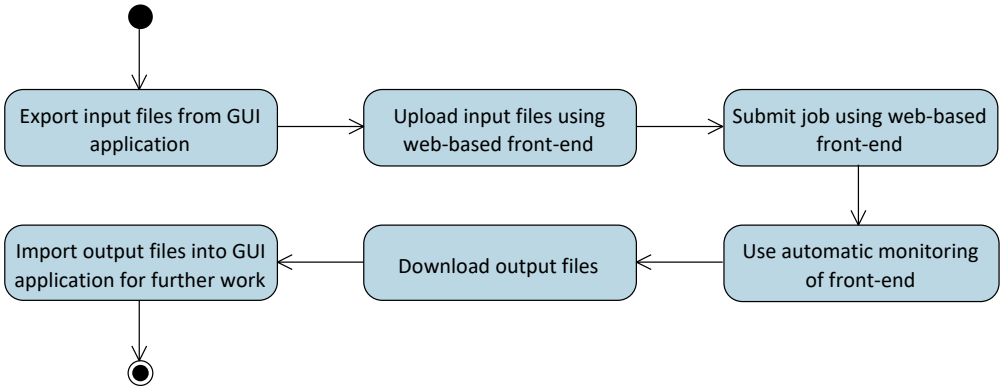

Figure 3.2: HPC cluster computing workflow using web-based front-ends. Web-based front-ends such as ChreME or Yabi circumvent the need for command-line operations but still require manual handling of input and output files.

to easily execute HPC workflows. Unfortunately, as visible from the public GitHub page [64], development activity has faded in recent months.

Even though Yabi offers a RESTful API that could in theory be used for integration into existing GUI applications, the customization possibilities are limited. As outlined in Section 4, the goal of providing transparent access to HPC resources from existing GUI applications requires highly customized web services for detailed progress information and flexible error-handling. It has to be possible to customize the API in order to meet the needs of specific GUI applications and computational jobs, such as LigandScout and virtual screening respectively.

3.2 Job Management System (JMS)

The Job Management System (JMS) introduced by Brown et al. [2] is one of the most comprehensive solutions. Similarly to the tools mentioned above, JMS allows users to submit jobs via a web-based front-end. It is then able to submit those jobs transparently to either a Torque or SLURM resource manager installation. Naturally, monitoring capabilities as well as features for managing a running job, such as suspending, resuming and cancelling a job, are also provided. JMS stores details for past jobs in a relational database, allowing users to examine past executions.

JMS differs from the previously mentioned CHreME and Moab HPC suite software through its support for building and executing complex computational pipelines. The web-based front-end allows users to combine tools, which can be any command-line utility installed on the HPC cluster, into multi-step workflows. Furthermore, it is even possible to upload new executable scripts directly through the web interface.

The downside that comes with the vast amount of features offered by JMS are its setup and configuration requirements. Even more concerning is the limited amount of support

available for the open source project. Similar to the Yabi project, the development has been stalled in recent months, which is derivable from the commit history of the public GitHub page [65].

3.3 Galaxy

The Galaxy project [66] is likely the most famous among attempts to connect scientists to HPC resources. It is an academic initiative, mainly developed by the Huck Institutes of the Life Sciences, the Institute for CyberScience at Penn State and the Johns Hopkins University, but it has been able to also draw support from a considerable open source developer community.

Similar to JMS, it offers a web-based front-end for submitting computational jobs. However, contrary to JMS, Galaxy is able to interface with all commonly used DRMSs, such as SGE, PBS, Torque, and SLURM. The possibilities for creating complex computational workflows using the front-end are even more extensive, allowing to make use of a vast amount of existing life-sciences tools.

The unique selling point of the Galaxy project is the possibility to share tools, workflows and data with collaborators. A further goal is to support reproducible research by capturing metadata about datasets, tools, and computational jobs [66]. Using this data, it should be possible to re-run experiments and validate results.

The focus of Galaxy is different from the one of this thesis. Galaxy does not aim to support existing GUI applications, but rather offers its own web-based front-end. There are no means for customizing the offered RESTful web services in a way, which would be required to provide transparent integration into existing applications. On the other hand, the support for workflow creation and resource sharing goes beyond the scope of this thesis. Additionally, the aim of Galaxy to support all kinds computational pipelines affects its web service interface, which offers only generic progress updates.

3.4 Taverna

In many aspects, the Taverna workflow suite [67, 68] is similar to Galaxy. Both focus on the bioinformatics community and provide means for composing and executing complex computational workflows. Nonetheless, the two projects have different goals. Taverna mainly aims for merging public web services into pipelines. Galaxy, on the other hand, facilitates working with locally installed software tools and can communicate with cluster resource management software. Taverna relies on plugins that expose resource manager functionality as web services in order to submit jobs to a HPC cluster. The range of functions provided by Taverna, which is a project supported by multiple research institutions and now part of the Apache Software Foundation (ASF), clearly exceeds the scope of this thesis. Fortunately, the target niche is different. Taverna can be used for web service orchestration and aims to be a general tool supporting a vast number

of services. As a result, the provided web-based front-end is extensive and complex. Monitoring of running workflows through the web-based interface is centered around viewing plain-text output of the respective services, while the goal of this work is to develop a transparent integration for an existing GUI application. Nevertheless, the web service orchestration capabilities of Taverna are of interest for this thesis, since they could be used to integrate the newly developed web services API, described in Chapter 5, into larger workflows.

3.5 Nextflow

An entirely different approach is taken by Tommaso et al. [69] with their Nextflow framework. This tool does not offer a graphical user interface and is therefore difficult to compare directly to the alternatives presented above. Nevertheless, Nextflow implements various interesting concepts that are of relevance for this thesis.

The main idea evolves around providing means for quickly building complex computational pipelines that can be submitted to a variety of resource managers. Nextflow requires the user to edit configuration files and submit jobs via the command line, but automatically handles parallelization by starting jobs that do not depend on each other in parallel. Similar to Galaxy, Nextflow also targets the fields of bioinformatics and life sciences in general, but it explicitly demands basic programming knowledge from its users.

As it neither provides a GUI nor web services that could be used for integration into existing interfaces, Nextflow is not an alternative to this work. It is however relevant for its capabilities to programmatically communicate with various DRMSs.

3.6 Remaining Research Questions

Even though various approaches and tools exist for enabling scientists to use HPC resources, multiple research questions remain. None of the existing tools are explicitly designed to be used for transparent integration into existing GUI applications. A concept that can vastly simplify the job submission workflow further, as illustrated in Figure 3.3. For this purpose, a customizable and extensible API for remote execution has to be available. In particular, detailed progress updates and information regarding occurred errors have to be ready for retrieval. Systems such as Yabi, JMS, and Galaxy are designed for a broad audience and aim to support the execution of any command-line tool. While this has obvious benefits, the necessary generalization affects the applicability to fine-tailored use cases. In general, their web service interfaces are mainly intended to be used by their own web-based user interfaces.

Furthermore, the actual integration into existing GUI applications is largely neglected in the scientific literature. Only some commercial software products offer the remote execution of large computational jobs. One prominent example is the AutoCAD computer-aided design (CAD) software [70] and its *Cloud Rendering* feature. However, this

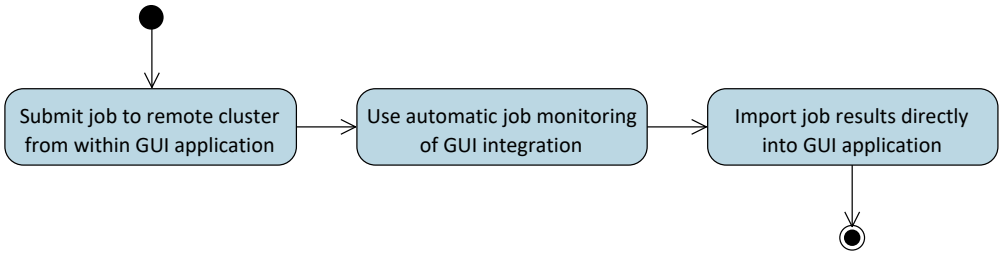

Figure 3.3: HPC cluster computing workflow using the proposed GUI integration. Integrating access to HPC clusters directly into the GUI applications used for everyday scientific tasks completely eliminates the need for manual input and output handling, resulting in a simpler workflow.

functionality does not allow the usage of an existing cluster, but rather demands the usage of Adobe cloud services that have to be paid for. Furthermore, progress updates are not directly incorporated into the AutoCAD GUI, instead users get notified via email when a job is finished. A more close example is the Maestro Molecular Modeling Software from Schrödinger [71]. It enables users to submit a variety of different computational tasks from the field of molecular biology to remote HPC clusters. The progress monitoring from within the GUI is limited to a display of the plain log file output from the remote command-line execution. Once again, the reason for this is that different tasks are handled and displayed the same way. Similar to Adobe, the Schrödinger company does not offer a public documentation or scientific literature regarding the used protocols and software architecture for its remote execution framework. The same applies for the integration into the Maestro GUI. Documenting best practices for this domain and formalizing common problems is therefore an open research question that is discussed in this thesis.

The support for existing DRMSs typically covers the submission of jobs, the monitoring of their progress to some extent, and the possibility of cancelling running jobs. However, a topic that neither existing web-based resource manager front-ends nor commercial desktop applications cover, is how to execute non-MPI applications on distributed-memory clusters. A possible approach is to split large computational jobs into multiple smaller sub-jobs, which can then be run on individual nodes. A software that integrates with DRMSs can aim to do this automatically and handle the process transparently to the user. None of the tools mentioned in this section provide this functionality. This likely stems from the wide range of tools these existing solutions aim to support. Splitting jobs requires in-depth consideration of the underlying process.

Additionally, existing solutions do not explicitly address the differences between physical on-site clusters and virtual cloud clusters. In order to exploit the benefits of HPC in the cloud, special considerations have to be made to account for elasticity and arbitrary horizontal scaling. An easy deployment process should further support dynamic cluster

allocations in the cloud. Of course, these additional features should not affect the usage with conventional on-site clusters, which are still the leading platform for scientific HPC.

The following two chapters present how this work tackles the described open questions. In particular, a newly developed software solution is described. Chapter 4 explains the architecture and design decisions that have been made in order to develop this software. Chapter 5 then details the concrete implementation.

CHAPTER 4

Software Architecture and Design

This chapter describes the most significant architectural patterns and design concepts that play a role in the software development process that is part of this work. We aim to conceive a general software architecture that allows for a transparent integration of remote services into GUI applications. Said services have to provide access to HPC clusters via interfacing with the installed DRMS. As a consequence, it should be possible to submit computational jobs to remote HPC clusters from within GUI applications without the need for any command-line interaction. This goal comes with a diverse set of challenges, partly introduced in Section 3.6. In particular, we can identify five main concerns:

1. Network communication with the remote HPC cluster and the accompanying challenges such as security risks and fault tolerance. In the following sections, the architecture arising from these challenges is referred to as remote execution architecture.
2. Integration and interaction with DRMSs.
3. Virtual screening on distributed-memory clusters.
4. GUI integration that allows for transparent invocations of the developed remote services.
5. Cloud computing considerations.

In the following sections, we describe these aspects in detail and propose solutions for each problem. Chapter 5 then presents the concrete implementation of our design.

4.1 Remote Execution Architecture

Before going into detail about the challenges that our remote execution architecture has to overcome, we can already present the first fundamental design decision. Clearly,

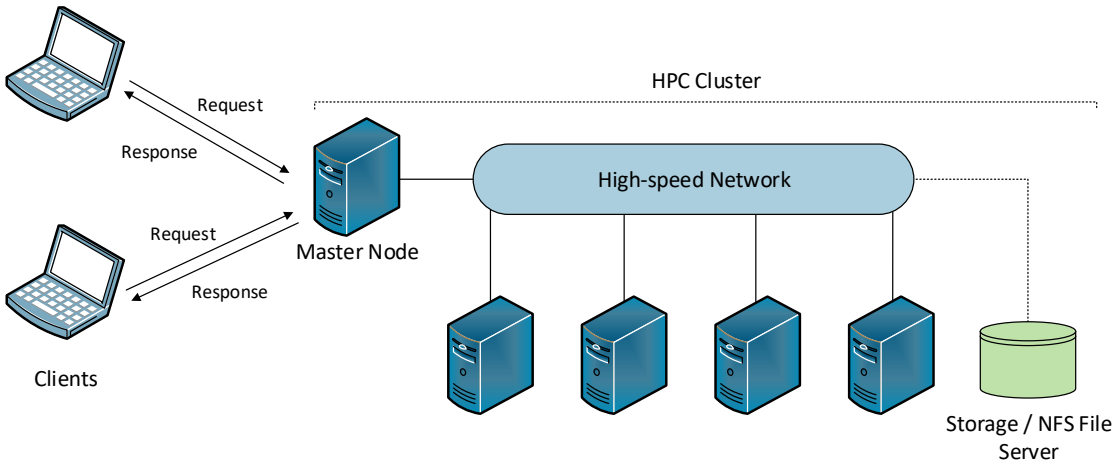

Figure 4.1: Client-server architecture with an HPC cluster as server. The master node of the HPC cluster also acts as host machine for the server application. This setup is especially useful when the master node does not act as compute node and therefore has unused computational resources.

the basic architecture of the system has to follow the classical client-server model. The equality of hosts, which would be essential for any peer-to-peer approach, is not present. The client-side is represented by a conventional desktop GUI application, whereas the server-side is a HPC cluster consisting of multiple nodes that appear as one resource to the client. Figure 4.1 illustrates this distributed systems architecture. The master node of the cluster acts as the host machine for the server application. This is not a necessity, but the obvious choice since it is usually not used for computation and therefore has free computational resources.

More interesting is the choice for the concrete communication protocols and related programming concepts. Over the past decades, various different approaches to realize distributed computing and facilitating inter-process communication across networks have been developed. Choosing the ideal technology for the use case at hand can be difficult due to conflicting goals. In order to analyze the most relevant concepts, we define the following criteria:

- **Fault Tolerance:** As the goal is to integrate access to remote HPC resources into an existing desktop application, the remaining functionality of the software must not be affected by errors concerning inter-process communication. In particular, a graceful degradation is important in regards to network related problems. Furthermore, the remote execution technology has to enable easy recovery from faults.
- **Interoperability:** LigandScout runs on all common operating systems, therefore, libraries providing support for the used technology have to be available for all platforms.

- **Ease of deployment** In order to facilitate easy access to HPC resources, the server-side application deployment has to be possible without extensive manual configuration.
- **Extensibility:** Ideally, the chosen communication technology can also be used to support future extensions to the desktop application. These can include incorporation of various third-party web services or direct access to cloud provider services.
- **Performance:** This factor resolves around the workload that is demanded from the server for fulfilling the requests of the clients and the bandwidth usage. The latter is dependent on the overhead of the used protocols.

Many of the above criteria imply a loose coupling of the client and server systems, which is why we consider this to be an important factor regarding our decision-making for the best suited technology. Loose coupling means, that the individual components need only little knowledge about the internals of the other system parts. As a consequence, components can more easily be replaced and updated without breaking the entire system. In contrast, errors in tightly coupled architectures are likely to affect multiple parts of the distributed application, having a negative impact on the requirement of fault tolerance. The degree of coupling also affects the ease of deployment. Complex, tightly coupled systems tend to require more manual configuration, because the individual components need to be finely tuned to work with each other. This hinders an automatic deployment process. Furthermore, tight coupling is preferably used within organizations where it is possible to exchange implementation details between hosts and establish a common middleware. This stands in stark contrast to public third-party services that have to be considered for future integration into the system, because these services provide documented APIs but do not expose implementation details. To summarize, loosely coupled systems can be considered superior when it comes to fault tolerance, ease of deployment, and extensibility. The performance criterion is an exception, as tight coupling typically comes with stateful protocols that are able to exchange resources in efficient binary formats. Therefore, tightly coupled distributed systems tend do achieve higher performance guarantees.

A classical remote procedure call (RPC) approach is an example for tight coupling. A client has to be aware of server internals, such as method names and parameter types. To facilitate communication, there is a rigid contract between client and server. Changes on either side (without updating also the other) necessarily lead to system failures. Furthermore, RPC is based on synchronous communication, which is generally considered to be a characteristic of tight coupling. As a consequence, in RPC-based approaches it is more difficult to achieve fault tolerance, ease of deployment, and extensibility. More complex remoting architectures such as the Common Object Request Broker Architecture (CORBA) [72, 73] come with similar disadvantages. Additionally, CORBA is in the process of becoming a legacy technology, which leads to decreased interoperability. All other technologies mentioned in this section are still widely supported across operating systems and programming languages.

Table 4.1: Requirements met by different remoting architectures. The valuations are only applicable for the use case at hand and should not be considered general evaluations of these technologies.

	RPC	**CORBA**	**SOAP**	**RESTful Web Services**
Fault Tolerance	✗	✗	✔	✔
Interoperability	✔	✗	✔	✔
Ease of Deployment	✗	✗	✔	✔
Extensibility	✗	✗	✗	✔
Performance	✔	✔	✗	✗

The remaining options are service-oriented architectures (SOA), specifically web services. The two most established variants are the Simple Object Access Protocol (SOAP) and RESTful web services. These are difficult to compare, as the former is technically a protocol and the latter is an architectural style. Nevertheless, they both denote web service concepts. The main benefit of SOAP is its extensive support for enterprise features. These include security-related protocols apart from SSL and reliable messaging implementations [74]. However, enterprise tooling support is of little relevance to this work. More important are the above defined criteria, some of which are difficult to meet with a SOAP-based implementation. Specifically, REST is favored when it comes to extensibility, because it has become a widespread standard for publicly available web service APIs. In the context of this thesis, important examples are the Amazon SimpleStorage Service API [75] and the ChEMBL web services API [76]. Both are candidates for future integration into the LigandScout GUI. Furthermore, SOAP services tend to require more bandwidth than REST-based web services, because message representation is limited to XML, which can cause an extensive overhead. However, both SOA variants can be expected to consume more bandwidth than complex middleware technologies, such as CORBA and RPC. One reason for this is, as already mentioned, the necessary parsing of text-based messages as compared to communication via binary object representations. In our use case, this difference can largely be neglected. The web service application is not expected to serve large amounts of clients, as it will be deployed within closed organizations.

Table 4.1 summarizes the benefits and disadvantages of the discussed technologies, though it should be noted that these are only valid for our particular use case. Taking the described factors into consideration, the decision has been made in favor of developing a RESTful web services API. An extensive description of the REST architectural style is given in the following section. We also explain in more detail how the unique attributes of REST aid our design.

4.1.1 Building upon RESTful Web Services

Representational State Transfer (REST) is generally considered an architectural style and not a protocol or technology. Giving a clear definition is not trivial, because it is composed of several architectural constraints of similar importance. They were initially specified in the doctoral thesis of Roy Fielding in 2000 [77]. Fielding defined REST in an abstract way, not bound to any particular protocol stack. However, RESTful web service communication now generally uses the HTTP protocol to access and manipulate online resources. The four most important principles are the following [74, 77]:

- **Resource identification through hierarchical URI scheme:** All resources exposed through a RESTful web service API have to be uniquely identified by a uniform resource identifier (URI) [78]. A URI is simply a string of characters designating exactly one resource. [79, p. 84].
- **Uniform Interface:** For all interaction between clients and servers, the standard HTTP verbs are used. First and foremost, these are PUT, GET, POST, and DELETE. PUT is used to create a new resource, which can later be removed using the DELETE operation. GET is invoked to access the current state of a resource. POST, in principle, updates an existing resource [74] or creates a subordinate resource [79, p. 84].
- **Statelessness:** Requests to a RESTful API happen in complete isolation, that means they are self-contained [79, p. 86]. The server expects the client to provide all information necessary to perform the requested operation within the request message. Stateful interactions mainly happen through the concept of connections between resources realized via hyperlinks [74]. In practice, a server often responds to a client's request with the URI denoting the updated or newly created resource.
- **Self-descriptive messages:** The resource representations available to clients are decoupled from the internal server representation [74]. Furthermore, the data format of a representation includes its media type and clients must request specific types. Other metadata can be included to detect errors, control caching, and handle security issues.

The resulting architectural style is very well suited for building a loosely coupled, distributed application that follows the client-server model. The client application only has to be aware of a set of URIs pointing to the resources that the server provides in its initial state. Resources that clients are allowed to access and that are only created during the ongoing operations of the server can be discovered through hyperlinks. Furthermore, implementing REST clients is straightforward, as only the HTTP protocol is used and a large amount of libraries exist for all platforms and major programming languages. Not using any special port or protocol, but instead relying solely on HTTP at the application layer and TCP at the transport layer of the OSI model [80] has the additional advantage of being compliant with most firewall settings. Also our proposed security and privacy

concept introduced in Section 4.1.2 profits from the fact that only one TCP port is required.

Due to the loosely coupled architecture and the stateless communication, fault tolerance and graceful degradation are achievable within RESTful applications. A key concept complementary to stateless operation is the granularity of services and their associated resources. Whenever a server is unable to fulfill a request for a specific resource, other requests for different resources can remain intact. As soon as the underlying problem is resolved, future independent requests for the initially demanded resource are served again. In the meantime, all functionalities of the GUI application that are not dependent on the troubled service endpoint can work as expected. An additional benefit of stateless service invocations is horizontal scalability. Because API requests are self-contained, multiple server instances can exist and answer any client request. As a consequence, the serving host no longer has to be a single point of failure. The proposed server application can run on any node of the HPC cluster that is able to submit jobs to the resource manager and that has access to all relevant files and directories. Unfortunately, it is left to the programmer to adhere to best practices and design the web service API in a fault-tolerant way. This is because the thin client-side libraries typically provide no respective functionality.

Chapter 5 documents the developed RESTful API in detail and explains what implementation work we have done in order to comply with the presented architectural principles.

Building upon the decision of using RESTful web services, several options are available for securing those services. All relevant approaches rely on TLS and HTTPS for securing the transport layer and guaranteeing privacy and data integrity of the communication [79, pp. 310]. For further analysis, it is helpful to have a separate view at authentication and authorization, the two parts that any security concept can be split into. Authentication takes care of identifying a user and confirming that they are who they claim. Authentication in RESTFul web services is usually done through HTTP headers [79, pp. 310]. The goal of the authentication process is to establish a common secret between client and server, which can be used in further requests for authorization. The responsibility of authorization is to determine whether an already identified user is allowed to perform a particular request.

The concrete protocols and technologies typically used for securing web services are not explained further here, because they are not relevant for this thesis. Instead, an alternative approach has been chosen. HPC clusters are rarely accessible from the public internet and even less frequent, any port other than the common SSH port 22 is open. This is because users of HPC resources are expected to have command-line experience and use SSH clients to access the target machine. Accordingly, requiring open ports for HTTP and HTTPS on the machine that hosts the server application is not feasible in this domain. The proposed solution is to delegate authentication to the SSH server that is expected to run on the host machine by setting up an SSH tunnel between the client and the server. Figure 4.2 shows how such a tunnel can be used to conduct secure web service

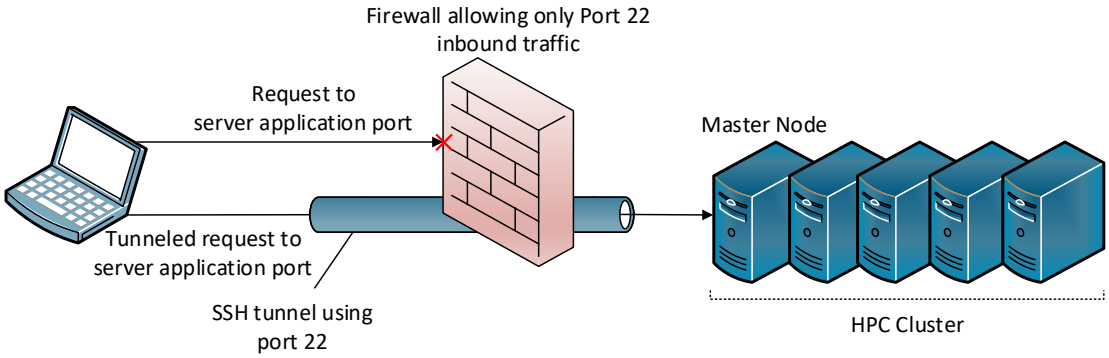

Figure 4.2: Accessing an HPC cluster through SSH tunneling. By use of a single SSH tunnel, any arbitrary port on a server can be reached, even if a restrictive firewall only allows inbound SSH traffic, which is often the case with organizational HPC clusters.

communication through port 22. This technique allows the server application to listen on an arbitrary port while no adjustments have to be made to the firewall configuration.

4.1.2 Security Concerns

The example shown in Figure 4.3 is more complex. Here, the HPC cluster is not accessible from the client network at all. Before being able to build an SSH tunnel to the cluster, an intermediate host has be used to forward the traffic to the cluster. To that end, two SSH tunnels are created, which subsequently enables the clients to reach the web services listening on an arbitrary server port. Once again, this does not require any specific firewall configuration. Both the firewall protecting the organization network as well as the firewall of the HPC cluster only need to allow inbound SSH traffic. For most other use cases, this approach would likely not be feasible, as users must already have an account on the serving host in order to access the deployed services. However, in our scenario, it can be expected that clients who are allowed to perform computations on a particular HPC clusters also possess accounts on the respective machines.

As one of the main goals of this work is to provide easy access to HPC resources, the users should not have to set up SSH tunnels manually before being able to communicate with the server application. Instead, the tunneling functionality is to be integrated directly into the GUI application, demanding only the input of user credentials.

The result of this solution is that the proposed server application does not have to perform authentication and authorization itself. Every user who is able to gain access to the port that the server application is listening to, necessarily has access to the underlying HPC cluster, otherwise they would not be able to construct the necessary SSH tunnel. In conclusion, the following benefits can be identified:

- **Simplicity:** Avoiding the need for security-related code allows for a simpler server application. This has positive effects on the maintainability of the software.

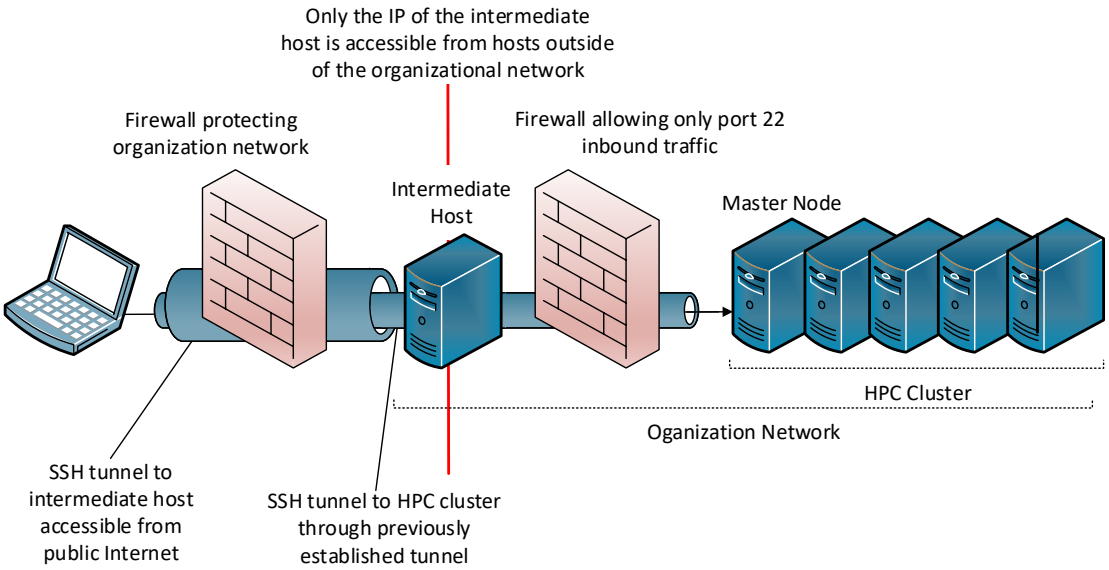

Figure 4.3: Advanced SSH tunnel setup through an intermediate host. If an HPC cluster is not only protected by a restrictive firewall but not reachable at all from the public internet, two SSH tunnels in combination with an intermediate host can be used.

- **Ease of Deployment:** The proposed concept aligns perfectly with common HPC system security policies. No adjustments regarding firewall configuration have to be made. Furthermore, existing accounts for terminal access can be reused to create SSH tunnels.
- **Proven Security:** Designing a system to be perfectly secure is complicated. RESTful web services usually have to deal with problems such as Cross-Site Request Forgery (CSRF) that are difficult to shut down completely, especially when dealing with complex attack vectors. On the other hand, the SSH protocol has been established for decades and has been proven to be reliable.

4.2 Interaction with DRMSs

The developed server application has to be able to submit jobs to the DRMS in use at the particular HPC cluster. Furthermore, it is required that specific jobs can be monitored and canceled. Unfortunately, these tasks are not trivial to design in a way that is applicable for all clusters. Section 2.2.2 outlined that there are various competing job scheduling and resource managers for cluster computing. The necessity to develop an interface for each system that should be supported follows as a logical consequence.

As mentioned in Section 2.2.2, this work focuses on SGE and its derivatives as the underlying resource manager. Nevertheless, the design has to account for future additions of other platforms. In order to comply with these requirements, the server application

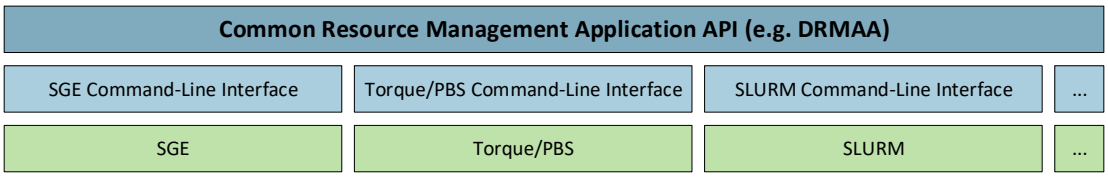

Figure 4.4: DRMAA as a unified way to access different cluster resource managers.

uses the *Distributed Resource Management Application API* (DRMAA) [31]. This API specification aims to provide a unified programming interface for cluster and grid computing software platforms, a concept visualized by Figure 4.4. The DRMAA specification was developed by the Open Grid Forum (OGF) in 2007 and has since been extended significantly. Unfortunately, DRMAA bindings do not exist for any combination of resource manager and programming language. However, in the context of this thesis, the Java language bindings for SGE and its derivatives are sufficient. They provide a thoroughly tested way to programmatically interact with HPC clusters that run SGE.

The alternative to using DRMAA would be to implement wrappers for the command-line interfaces of the different resource managers. This is a trivial, but tedious task. The required output parsing is predestined to be error-prone, since each of the different resource managers uses its own output format and, for example, has its own set of error messages that must be recognized correctly. Nevertheless, this is a task that may be required in the future in order to support cluster computing platforms that do not offer DRMAA Java bindings. Nextflow [81], a project already presented in Section 3.5, does not depend on DRMAA but instead uses its own custom wrappers for programmatic access to various resource managers. Hence, the Nextflow project shows that this is also a viable approach.

4.3 Enabling Virtual Screening on Distributed-Memory Clusters

Section 2.1.2 explained that HPC clusters are distributed-memory computers that require a different programming model than regular shared-memory machines. In order to make use of multiple cluster nodes, application developers have to specify how data should be distributed over the memories and when it needs to be moved for inter-node communication. One way of doing this is the development of MPI applications, which can then be executed on multiple cluster nodes at once. In this work, we take a different approach for various reasons. First, the LigandScout virtual screening command-line tool, which we use for executing VS jobs, is not currently implemented as an MPI program and doing so is not part of this thesis. However, there are other, more important reasons. Relying on the executed application to take care of spreading over the complete underlying HPC cluster would eliminate the ability to also support non-MPI applications. Furthermore, running MPI jobs requires beforehand planning regarding how many cluster

nodes are to be used. In practice, this is often done manually by simply communicating with the personnel in charge of cluster administration or by using as much resources as possible. The latter leads to problems already explained in Section 2.2.2. In particular, long jobs that use large parts of a cluster can block short jobs that would just require few nodes. This would be even more problematic in the case of a server application, which has to determine the number of nodes to use in an automatic way and for a potentially large amount of jobs.

Instead of delegating the handling of the distributed memory to the executed applications, the server application developed in the course of this work takes care of splitting jobs into sizes that can be executed on individual nodes. The result is a set of shared-memory sub-jobs that can be run independently and therefore without inter-node communication. Figure 4.5 gives an abstract visualization of this concept. In the proposed solution, the remote virtual screening process essentially starts by determining the compound database size. Then, an arbitrary amount of VS sub-jobs are submitted to the DRMS, depending on the amount of input compounds to be screened and a number of parameters that are introduced later in this section. The DRMS is then free to schedule these jobs across all available cluster nodes. All sub-jobs run independently and write their progress information and their results to files on the shared cluster storage. Naturally, these results of the individual sub-jobs have to be aggregated and returned to the client as a whole. Fortunately, virtual screening hits are independent and can easily be merged from different files. The benefits of the proposed solution are manifold:

- **Flexibility:** Jobs can be split into arbitrary sizes, which means that the approach works for all cluster and node sizes.
- **Integration with Resource Manager Policies:** Using multiple relatively small jobs allows DRMSs to apply fine-grained scheduling policies and prioritization. This may be necessary to comply with organizational policies.
- **Support for non-MPI Tools:** The custom job-splitting mechanism also supports tools that are not developed as distributed-memory applications. This is the case for virtual screening, but also for many other scientific HPC applications, which are often not developed by professional programmers.
- **Low Latency:** Small jobs are more likely to be run immediately, because the chance that enough slots are available is higher. As a consequence, first progress updates and intermediate results can sooner be delivered to the user if jobs are split into multiple smaller sub-jobs. This is a major benefit for this work, because users of GUI applications expect to be shown almost instant visual feedback.

However, the approach comes with two drawbacks. Splitting jobs and aggregating results demands computational resources from the node running the server application. Section 5.2 presents the implementation choices made in order to keep this demand as low as possible. Furthermore, starting multiple sub-jobs instead of just one large job

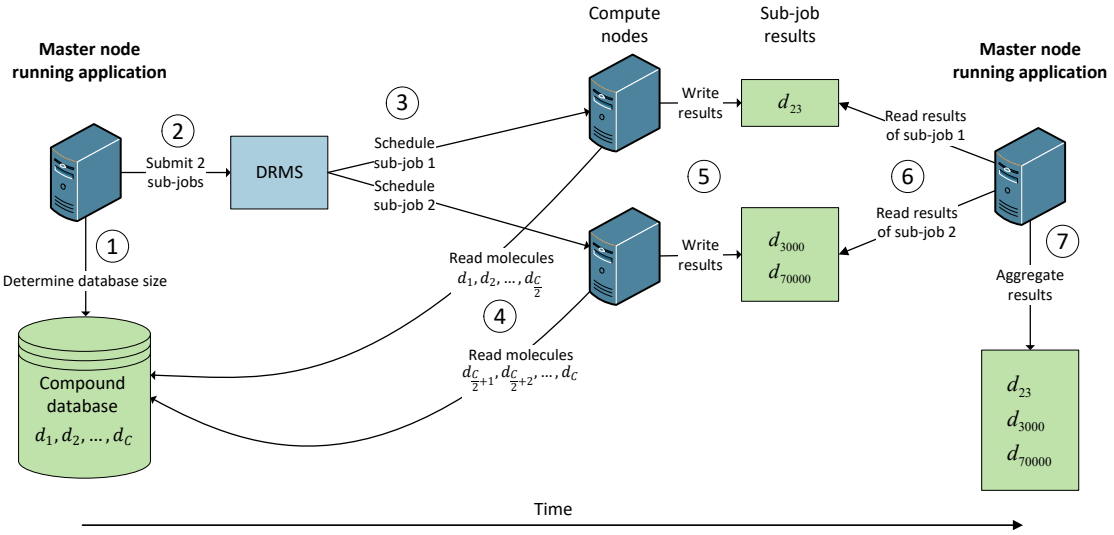

Figure 4.5: Splitting of a virtual screening job into two smaller sub-jobs. Each of the two sub-jobs is responsible for screening half of the compounds present in a particular database. The two machines titled *Master Node* are the same and drawn twice only for visualization purposes.

comes with computational overhead both for the resource manager, which has to apply scheduling algorithms, as well as for the executed application. The latter typically starts by performing initialization tasks, such as opening input streams, that would otherwise have to be done only once. Section 6.1.3 provides a detailed analysis on the performance implications of the developed algorithm.

In order to provide a terminological foundation to these later sections, it is necessary to introduce some terms and concepts and also to precisely define already introduced ones:

Job: In this thesis, a job describes a complete virtual screening experiment. The job can be submitted to the DRMS as a whole or splitted into multiple sub-jobs.

Sub-job The sub-job term refers to a computational task that can be performed on a single cluster node. A single job is split into an arbitrary amount of these tasks, depending on the sub-job size.

Sub-job Size (z)**:** The notion of size in this context means the number of compounds that have to be screened by a single sub-job. The number of query pharmacophores also affects the runtime of sub-jobs, but is not considered by the job-splitting algorithm. This is because every pharmacophore has to be checked against every database compound, hence only one of the two input types can be accounted for when generating sub-jobs. For any given virtual screening experiment with its total number of input compounds, the amount of generated sub-jobs is inversely

proportional to the sub-job size. We introduce the variable z to denote the sub-job size.

Slot Consumption (u)**:** Another important notion is the slot consumption of a sub-job. We introduce this term as the amount of resource manager slots, that every single sub-job occupies. In the context of this work, resource managers slots always imply CPU cores. The maximum slot consumption value that is possible on a given cluster is therefore equal to the amount of CPU cores a single node provides. In the following, we will use the variable u to denote the slot consumption.

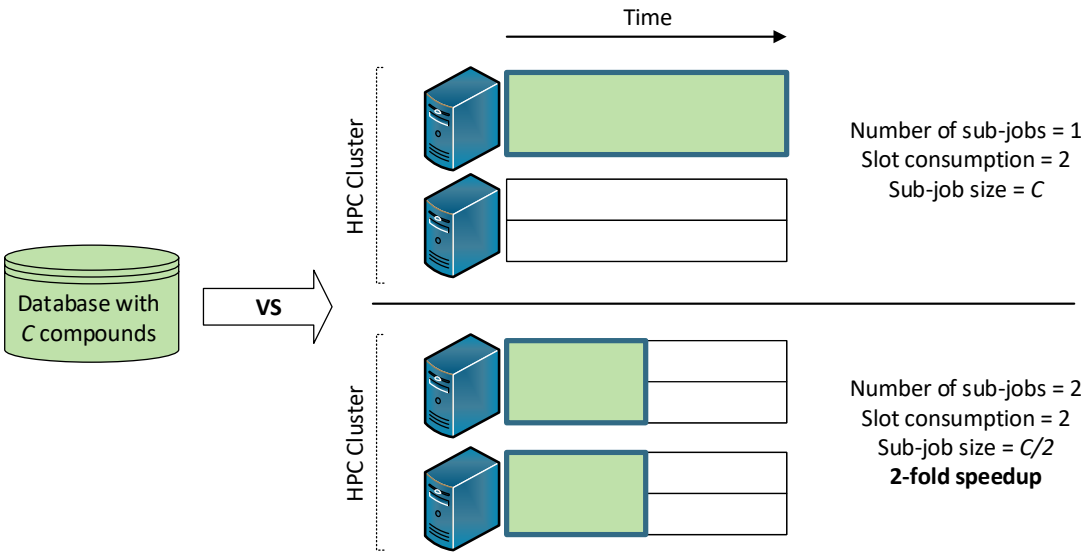

Figure 4.6: Exploitation of multiple distributed-memory nodes via job-splitting. The top row illustrates the naive approach of running a VS job on only one of the available cluster nodes. In the bottom, the job is split into two equally sized sub-jobs that can be run in parallel on two separate nodes. The exemplary HPC cluster contains two nodes that each offer two slots.

In Figure 4.6, it is shown how the sub-job size and slot consumption parameters can be used to distribute a virtual screening job across multiple cluster nodes. Without splitting the job, it can only run on one of the nodes of the distributed-memory cluster, while the other nodes remain idle. By splitting the job into sub-jobs, all cluster resources can be exploited. Using the described parameters, it is further possible to exert fine control over how many sub-jobs are started and also over how those sub-jobs behave in the cluster environment. A more detailed look into this is given in Section 5.2, where we discuss the concrete implementation of the presented approach.

4.4 Designing a Graphical User Interface (GUI) Integration

One of the major goals of this work is achieving transparency for the user regarding the internals of the client-server communication and the interaction with the DRMS of the target HPC cluster. Therefore, the integration into the graphical interface must not look like yet another resource manager front-end. On the other hand, it should be possible from within the GUI desktop application to perform typical resource manager functions, such as monitoring and cancelling jobs.

The proposed solution for virtual screening in LigandScout provides two ways for executing a remote screening on an HPC cluster.

1. **Full Transparency of Remote Execution:** This variant is intended to behave exactly as if the screening would be performed on the local machine. The network communication that is performed in the background as well as the used DRMS functionality should be completely transparent to the user. Therefore, no detailed information about the status of specific sub-jobs is displayed, instead the progress updates are aggregated and presented to the user the same way as they would be during a local screening.
2. **Advanced, Asynchronous Execution and Monitoring:** In this case, the remote screening job is started on the HPC cluster, but the GUI does not wait for progress updates and allows the user to continue local work. Naturally, this approach requires an additional GUI representation for job monitoring and management. In practice, this will be a dialog that shows all current and past remote jobs along with progress information. Users should then be able to download the data corresponding to the listed virtual screening experiments, such as input pharmacophores and hit compounds, and also to cancel or delete specific jobs.

The design further allows users to switch between these variants. In particular, the blocking progress monitor windows that comes with variant one can be disabled at any time, resulting in a seamless transition to variant two. In both cases, the underlying client-server communication has to happen transparently. This requires efficient data transfer mechanics in order to guarantee short response-times. Another challenge is the seamless integration into existing GUI parts, which must also ensure that users who do not use the developed features are not affected by their addition.

4.5 Cloud Computing Considerations

HPC clusters in the cloud differ from on-site clusters in a number of ways. As the developed server application should integrate well with cloud deployments, these differences have to be considered and accounted for. In practice, one major issue is that cloud clusters are not within the network of a given organization. This has significant security implications and is a substantial issue especially for companies that do research in the life sciences

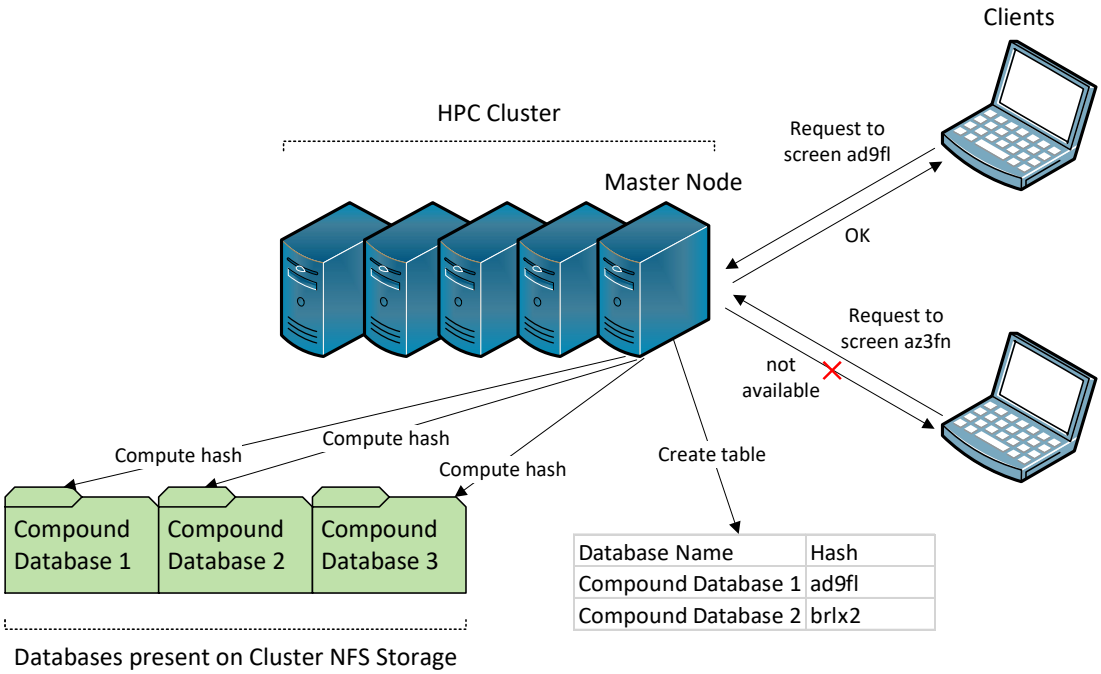

Figure 4.7: Database identification via hashes. The server-side hashing of databases allows clients to submit screening jobs without transmitting potentially large compound libraries over the network.

area, because drug candidates are valuable intellectual property. Fortunately, the security concept relying on secure SSH tunnels presented in Section 4.1.2 meets all concerns by using a well established protocol in order to perform authentication and guarantee data integrity and privacy.

Another consequence of the changed topology is a slower network speed. In the case of virtual screening, this mainly implies that it is not feasible to transfer large input compound databases for every screening job. On the other hand, to implement a transparent GUI integration, users should be allowed to select local databases as an input for remote screening jobs the same way as they would do before local execution. The proposed solution accomplishes this by only sending hashes of the input databases with every screening request. Hashing can be conducted by any cryptographic hash function, such as MD5 [82], which is commonly used for identifying files also in the field of digital forensics [83]. The actual compound databases are stored permanently on a storage device accessible to the HPC cluster. Figure 4.7 illustrates this process. Needless to say, this approach requires appropriate error handling in case one of the requested databases is not actually present on the storage of the cluster. Also, a separate dialog must be provided for selecting databases that are not locally available on the client machine. Section 5.1.2 details the concrete implementation of this feature.

The developed software should not only work with cloud deployments but harness their full potential. Therefore, it is critical that the server application can make use of the scalability and elasticity benefits of cloud clusters, which were described in Section 2.3. Fortunately, the job-splitting approach presented in Section 4.3 provides the required flexibility. Jobs can be split into arbitrary sizes, which allows to make use of huge cloud clusters. At the same time, larger job sizes can support vertical scaling, for example the largest instance type of Amazon EC2 comprises 64 virtual CPU cores [84]. The synergy with the elasticity qualities of cloud deployments is even more evident. Large jobs that are split into multiple smaller ones can be started on clusters of any size and also make use of later dynamic addition of additional nodes by increasing the number of sub-jobs running in parallel. Consider the example of a small cluster and a large screening job that is split into many sub-jobs, each planned to run on all slots of a single node. Some of the sub-jobs can immediately start to execute while additional nodes are added to

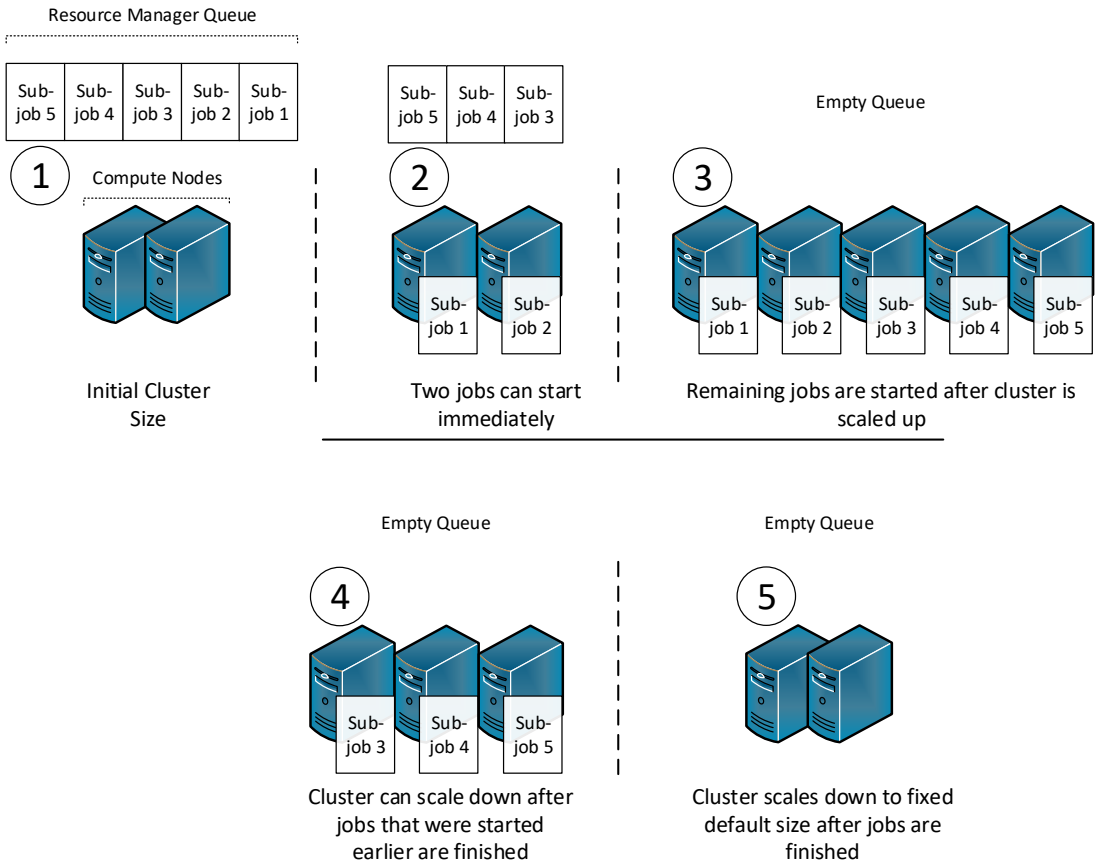

Figure 4.8: Automatic cloud cluster scaling due to pending jobs. If more sub-jobs are submitted than the current amount of nodes can execute at once, the elasticity of cloud deployments enables automatic upscaling. After the jobs are finished, the cluster is scaled down again.

the cluster. These can then be used to execute jobs that are waiting in the resource manager's queue. Once the number of jobs becomes less than the number of nodes, the cluster can start to scale down and reduce running costs. Figure 4.8 shows a concrete example with five sub-jobs submitted to a cluster with an initial size of two compute nodes. This way of exploiting elasticity would not be possible if distributed-memory computing compatibility would be provided through an MPI-enabled virtual screening implementation.

The last significant challenge in this context is to enable a fully automated deployment process. This is necessary in order to support on-demand cloud clusters that could potentially be purchased only for the duration of single virtual screening experiments. It should not be necessary to install software, such as the developed server application, manually on the cloud cluster after the corresponding instances are allocated. Fortunately, the desired automated deployment process is enabled by tools such as CfnCluster. Refer to Section 2.3.3 for an introduction to the capabilities of said toolkits. CfnCluster allows to specify a so-called *post install script* that is executed on every node of the cloud cluster. This feature can be used to install required software and establish a specific folder structure. However, it is not possible to provide manual input while the script is running, meaning that the procedure must be able to run autonomous. The chosen approach for this work is to deploy a self-contained server executable, able to run with a provided default configuration. To simplify the process further, an embedded web-server as well as an embedded SQL database are used by default. Section 5.4 explains how these approaches allow for a ready-to-use cloud deployment using only a single CfnCluster command.

CHAPTER 5

Implementation of the Distributed Virtual Screening Architecture

This chapter presents the concrete implementation of the software developed in the course of this work and details the most significant technology choices. It is further shown how the abstract concepts and decisions introduced in the previous chapter are reflected in the implementation.

5.1 Job Submission and Controlling Server

The server application is the central part of the developed solution. It communicates with clients, handles interaction with the underlying cluster resource manager, and manages persistent storage of information associated with computational jobs. The software is wholly written in Java and developed using the Java Development Kit (JDK) 8u151. This is partly because LigandScout is also a Java application and homogeneity of programming languages comes with obvious benefits. Additionally, Java is a mature language with extensive support for web service development and enterprise server features. The second most important technology decision concerns the Spring Framework [85, 86]. Setup and configuration of the framework is facilitated by Spring Boot [87]. As the following sections point out, the open-source application framework and its accompanying technology stack are used for all major aspects of the server program. The version numbers of all used frameworks and libaries are listed in Appendix A.

In order to comply with the the famous principle *Keep it Simple, Stupid* (KISS) principle, a straightforward layered architecture has been developed. This integrates well with the used Spring Framework. The three layers, repository layer, service layer, and controller

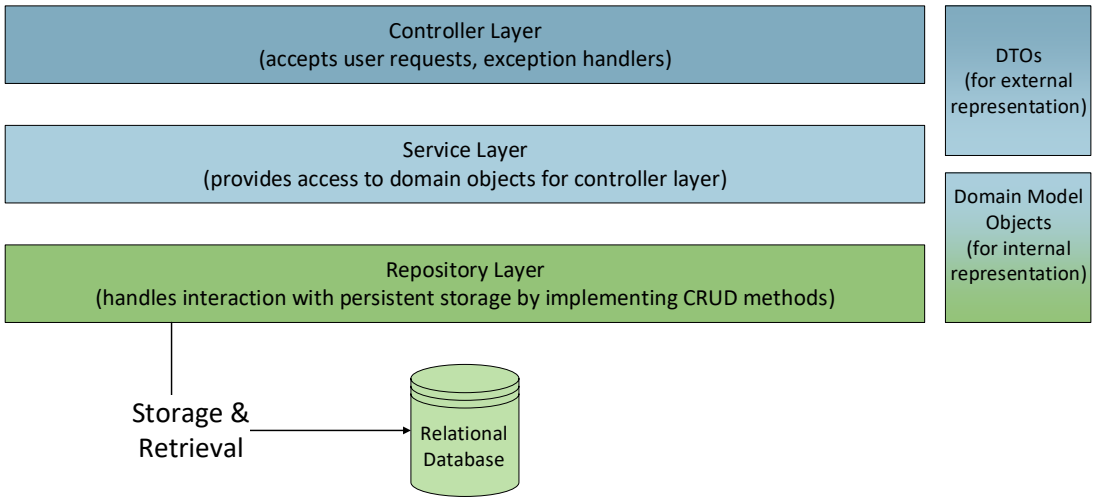

Figure 5.1: Layered architecture of the server application software.

layer are shown in Figure 5.1 and described throughout this chapter. The application further follows a domain-driven design (DDD) [88, pp. 3-4], hence, the database entities are internally represented by domain model objects. As visible in Figure 5.1, these objects are used in the repository and service layers. The controller layer, on the other hand, relies on data transfer objects (DTO) in order to accomplish a clean separation between internal and outside representation. Section 5.1.3 explains how the DTOs are used to generate JSON representations that can be served to clients. The separation into DTOs and internal domain model objects also has the benefit of being able to change one of the two without affecting the other. In practice, the internal implementation often changes, while the outside representation that clients depend on, has to stay the same.

In the following, we will look at each of the three aforementioned layers individually.

5.1.1 Repository Layer

The server application relies on a relational database for persistent storage. Access to this database is given by the repository layer to the above located service layer through create, read, update, and delete (CRUD) operations. The choice for a specific database management system is left to the user and can be configured through the externalized configuration concept of the Spring Framework. Other possible configuration options are described in Section 5.1.2. In order to facilitate the programmatic database access Hibernate and Spring Data are used. Once again, the exact version numbers are given in Appendix A. The combination of these technologies allows for clean and simple CRUD code that maps database entities directly to internally used domain model objects, a concept known as object-relational mapping (ORM). Figure 5.2 shows the database schema as entity relationship (ER) model. A part of the resulting class diagram of domain model objects is depicted in Figure 5.3, which also shows that the computational

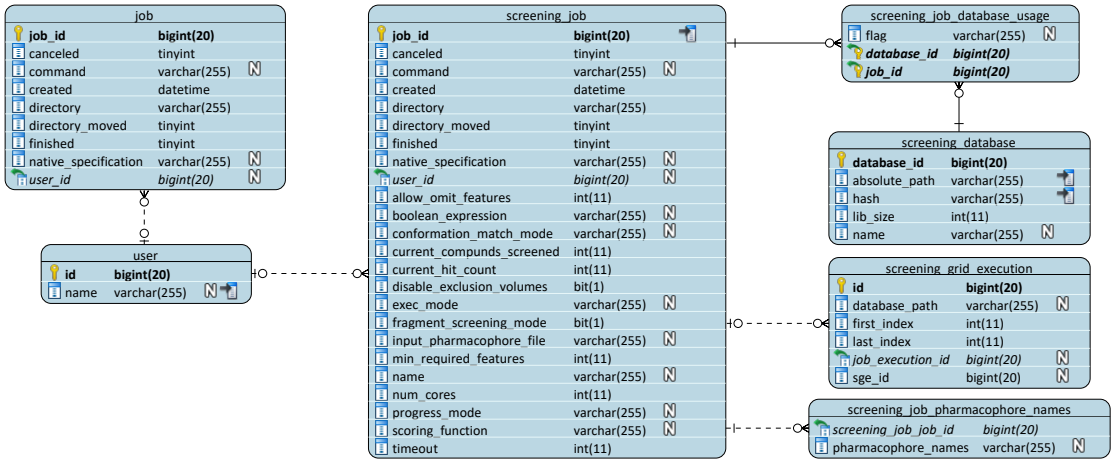

Figure 5.2: Database scheme. The ER-Diagram visualization shows the server application's persistent storage entities and the columns of the corresponding relational database tables.

jobs are implemented in a hierarchical manner in order to support future additions of computational job types. The fields common to all jobs can be derived from both the `job` table in Figure 5.2 and the `Job` class in Figure 5.3. The same goes for the specialized `ScreeningJob` entity. The class diagram for all entities of the ORM is provided in Appendix B.

In short, the relational database stores information about all submitted jobs. This is, of course, necessary to be able to answer subsequent requests for the status or results of a specific job. The actual results of screening jobs are thereby not stored in the database, but instead written to separate files directly by the executed VS command-line tool. Section 5.1.2 outlines how the server application keeps track of these files and exploits their content. However, Figure 5.3 already shows that the domain model objects and therefore also the corresponding database table contain information about the location of files related to a specific job.

Since ease of deployment is a major concern, also in regards to cloud deployments, an embedded H2 database is set up by default. As a result, users can choose to skip any database-related configuration and use the provided application out of the box.

5.1.2 Service Layer

The service layer implements the business logic that is needed to perform the controller's requests. As such, it is the most interesting to analyze. While the implementation specifics of the other layers are largely predefined by the technology decisions, the service layer contains original algorithms and logic.

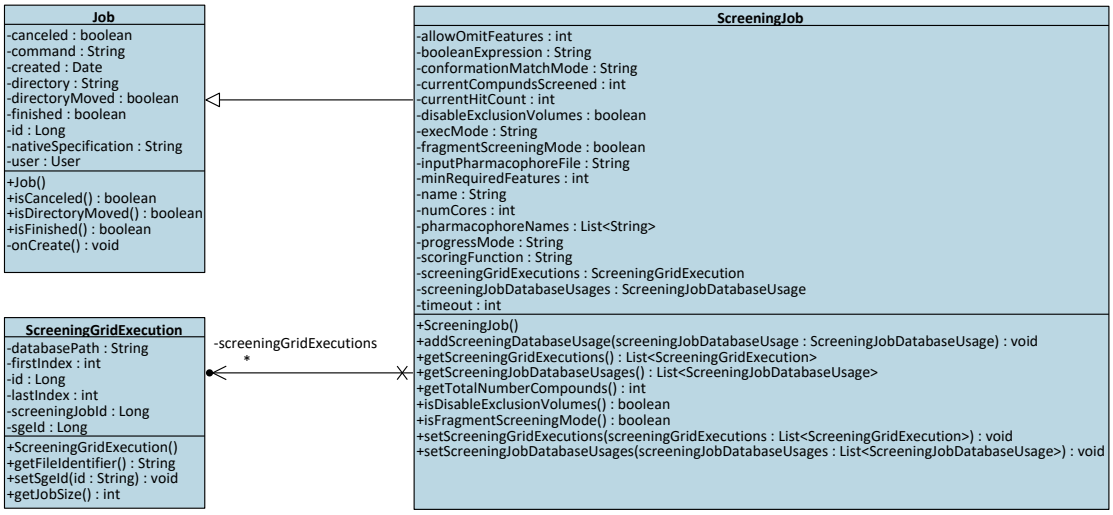

Figure 5.3: `ScreeningJob` class diagram. The most relevant database entities for screening jobs are shown. These are the more general `Job` superclass and the `ScreeningGridExecution` class, which is used to hold information about individual sub-jobs.

Configuration Management

The developed server application provides a variety of configuration options that can be set via a text file. This configuration file can be located either in the same folder as the server executable or in a dedicated `/config` subdirectory of the current working directory. If both are present, the latter option has the higher priority, which allows different users to easily start the server application with their own configuration. We have implemented this approach by use of Spring's externalized configuration mechanism. Table 5.1 shows the most important available settings and summarizes their purpose. The parameters `sge.amount.slots` and `job.splitting.chunk.size`, which concern the developed job-splitting algorithm, are especially important in the context of this thesis. Many of the options can be overridden within client requests. Within the service layer, a singleton bean is responsible for reading the settings file and providing the read values for other classes. For this purpose, the bean can be injected using Spring's dependency injection mechanism. Appendix C contains the full default configuration file.

Hashing of Compound Databases

The implementation of the approach already presented in Section 4.5 and illustrated in Figure 4.7 is not completely straightforward. Compound databases, as they are used by the LigandScout virtual screening command-line tool, are essentially relational databases of arbitrary size. As a consequence, hashing complete databases is not feasible, especially given that many libraries may be given and regular refreshes are needed. A first step for reducing hashing time is the decision in favor of the MD5 hashing algorithm, which

Table 5.1: Selection of the most relevant server application configuration options

Settings Key	Description
`server.port`	Port that the server application should listen on.
`job.directory`	Folder location for saving files related to computational jobs.
`iscreen.path.executable`	Path to the LigandScout virtual screening command-line tool.
`libsize.path.executable`	Path to the LigandScout command-line tool that is used for determing the size of a compound library.
`ldb.directories`	Comma-separated list of directory paths, that contain compound databases. The databases found in the given directories are made available for screening.
`ldb.directories.recursive`	Flag to specify whether the directories given with `ldb.directories` should be searched recursively.
`sge.parallel.environment`	Parallel environment to be used for submitting jobs to SGE.
`sge.amount.slots`	Number of SGE slots to use per sub-job i.e. slot consumption.
`sge.priority`	SGE priority value to be used for submitting jobs.
`sge.option.string`	Additional options to be transmitted to SGE when submitting jobs.
`iscreen.amount.cores`	Number of threads to be started by the LigandScout virtual screening command-line tool. It can sometimes be beneficial to set this higher than `sge.amount.slots`.
`iscreen.memory`	Amount of memory in gigabytes (GB) to be used by the LigandScout virtual screening-command-line tool.
`job.splitting.chunk.size`	Maximum number of compounds that each sub-job should screen i.e. sub-job size.
`move.finished.jobs`	Flag that turns on automatic moving of finished jobs to a directory different from the one given by `job.directory`.
`finished.jobs.directory`	If `move.finished.jobs` is set, finished jobs are moved to this directory.

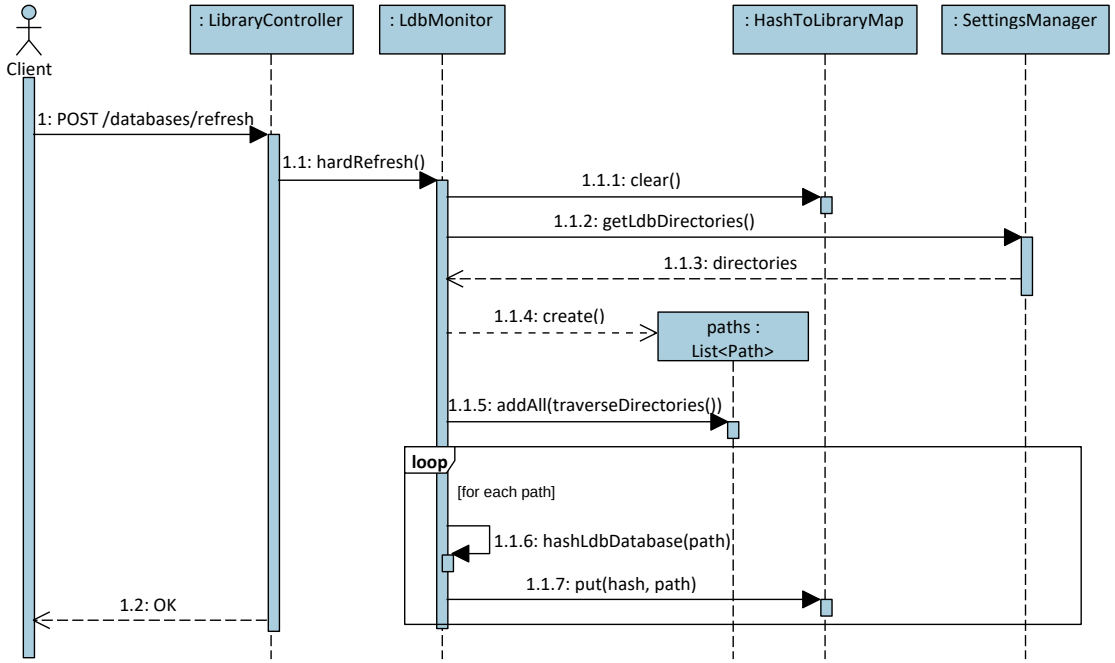

Figure 5.4: Sequence diagram of the server-side hash to path mapping creation. The mapping is not only refreshed upon explicit client request, but also at application start and when an unknown library is requested. The diagram omits several utility methods for visualization purposes.

is easier to compute than more advanced algorithms such as SHA-2 or SHA-3. The infamous shortcomings of MD5 [89] do not matter in this context, since the purpose of file identification is not related to security.

However, even with MD5, it is not possible to handle an arbitrary amount of compound libraries in reasonable time. Instead, our implementation hashes only the transaction log files of the relational Derby databases. The computation of these hashes can be done in negligible time even for very large amounts of databases. Tests with hundreds of compound libraries have still completed within less than one second. A Derby database logs all committed transactions and keeps the log file small through periodic checkpoints [90]. The combination of checkpoints and transaction log data is enough to uniquely identify a relational database.

Figure 5.4 illustrates the workflow that the server application performs in order to establish a mapping from compound database hashes to the respective file paths. Hashes are refreshed either upon an explicit client request or when a client requests screening of a database that is currently unknown to the server application. In the latter case, the server refreshes the database list and only returns an error message when this process still does not reveal the database in question. Doing periodic database refreshes would consume computational resources but not solve the problem completely. Even if automatic

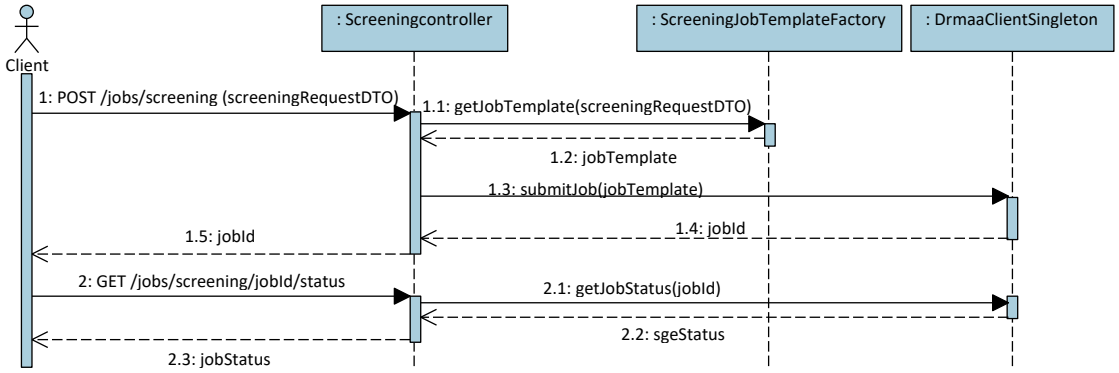

Figure 5.5: Sequence diagram of the job submission process. The depiction is heavily simplified. Operations related to persistent storage are omitted. In practice, the SGE ID is not used for job identification, instead an internal ID correlated to the database table identifier is used. Furthermore, most status information is parsed from a job's output files instead of directly retrieved from SGE.

refreshes would happen very frequently, unfortunately timed requests could still concern databases that were added just after the last refresh operation.

DRMS Access

Interaction with the SGE cluster resource manager is realized by use of Java DRMAA bindings, as already explained in Section 4.2. Concretely, a singleton is implemented that provides a DRMAA session, which can then be used to send commands to the DRMS.

Job submitting is the most important and most complex task that is implemented in this context. The Java DRMAA client library expects a `JobTemplate` object, that contains all relevant options and paths necessary to start the execution of the external LigandScout VS command-line tool. The service layer therefore provides a factory class that can be used to construct such objects. The factory also takes care of applying the configuration options introduced in Section 5.1.2. After submission, DRMAA returns the ID of the newly created job. This value is saved into the database and then used for monitoring and management purposes. Figure 5.5 illustrates the workflow of job submission and subsequent monitoring. Job types other than virtual screening can easily be supported by implementing further factory classes that are capable of constructing the respective `JobTemplate` objects.

File Management

The management of input, output, and log files of individual computational jobs is an important task of the server application. For each screening job, a number of files have to be stored on a file management system available to the host machine. Table 5.2 lists these files and also provides information on whether the respective file is present once per

Table 5.2: Files that are created for each screening job

File	Description	For each sub-job
`errors-and-progress-report.txt`	Progress information and error log	yes
`non-error-log.txt`	General log providing job summary	yes
`result-hitlist.sdf`	Result file containing the hit compounds	yes

screening job or created for each sub-job. Section 4.3 explained the concept of sub-jobs and how they are used to fully utilize HPC clusters. The concrete implementation of this approach is detailed in Section 5.2. In any case, the application provides a configuration option to specify the base location for all created files (cf. Table 5.1) and also lets users request the deletion of all files related to a specific job in order to reduce storage consumption.

The presence of one progress file for each sub-job, hence, multiple files for each job, comes with the need for result aggregation. Upon a client's request for the current status of a specific job, all progress files that belong to the respective job are parsed and their progress is accumulated. In order to make this process as efficient as possible, the parsing logic only considers the last couple of lines of each relevant file. This is enough to retrieve the number of already screened molecules along with the current hit count. In order to further speed up status retrieval requests, the necessary information for already finished jobs is saved to the relational database and not gathered again through parsing upon subsequent client requests. Parsing is also omitted for jobs that are still waiting in the queue of the DRMS.

Testing at the Department of Pharmaceutical Chemistry of the University of Vienna has brought to light an additional necessary feature. HPC clusters often use different storage volumes that are connected to networks of different speeds. These different storage options serve distinct purposes. In particular, file locations connected to the high-speed network of a HPC cluster are often not provisioned for long-term storage. In order to facilitate working with such a setup, the server application is capable of automatically moving files associated to already finished jobs to a different folder that can be specified in the settings file, as can be seen in Table 5.1. Currently, this cleanup process happens every full hour. Naturally, it is essential to keep track of where the files corresponding to a specific job are currently located. Otherwise, subsequent user requests for the status and results of this job could no longer be answered. Therefore, the location of the directory that holds all files for a particular job is persisted in the relational database.

Table 5.3: Web service resource endpoints exposed by the server application

HTTP Verb	URI	Request Parameters
GET	/status	
POST	/jobs/screening	
GET	/jobs/screening/submissionDetails	
GET	/jobs/screening/<jobId>/status	
GET	/jobs/screening/<jobId>/hitlist/	
GET	/jobs/screening/<jobId>/pharmacophores/	
GET	/jobs/screening/<jobId>/session	
POST	/jobs/screening/<jobId>/cancel	
GET	/jobs/screening/<userName>/statuses?	pharmacophoreName databaseName
DELETE	/jobs/screening/<jobId>	
GET	/databases	
GET	/databases/<hash>/availability	

5.1.3 Controller Layer

The controller layer is responsible for the communication with external clients. For this purpose, it implements a RESTful web service API. This layer also has to handle any exceptions and errors that either originate as internal server errors in the below layers or are caused by invalid user input.

The RESTful web service API is a fundamental part of the server application and as such is planned most thoroughly. Furthermore, clients are dependent on the API to follow a documented structure. Changes to the externally exposed services also have to be considered within client implementations and are therefore costly to implement. The various resource endpoints are listed in Table 5.3. It can be seen that the URI scheme and usage of HTTP verbs are designed to comply with the architectural principles presented in Section 4.1.1. The `/status` resource provides trivial information about the state of the server application and can be used to test its availability. The second group of resources, which uses the common prefix `/jobs/screening`, focuses on computational screening jobs. By making use of the descriptive HTTP verbs, new jobs can be submitted, various data related to specific existing jobs can be retrieved, on-going executions can be cancelled, and jobs can also be deleted from the database along with all related output and log files in order to free server resources. As is indicated by the third column of Table 5.3, the API also provides filtering capabilities for screening jobs by name of a used pharmacophore model, a used database or both.

Currently, all resource endpoints provide JSON as response format. Listing 5.1 shows an exemplary response of a call to the `jobs/screening/<jobId>/status` resource endpoint. Responses of the other services are exemplified in Appendix D. It can be seen that the status of a screening job contains progress information as well as the settings and input data that were used to start the screening run. This plays an important role for the GUI integration detailed in Section 5.3, since users have to be able to access this information. The list of objects given as `sgeExecutions` presents information about all individual sub-jobs submitted to the DRMS. Further information about the implementation of this feature is given in Section 5.2.

Listing 5.1: Exemplary response of a call to the `/jobs/screening/<jobId>/status` web service resource endpoint.

```
{
    "compoundsScreened": 56874,
    "created": 1507812811567,
    "currentHitCount": 3,
    "errorMessage": "",
    "finished": false,
    "internadId": 148,
    "name": "job_148",
    "pharmacophoreNames": [
        "(1KE7) [A] LS3299",
        "(1KE5) [A] LS1299",
        "(1KE6) [A] LS2299"
    ],
    "screeningDatabases": [
        "ZBC_13062016_filtered_chunk1.ldb",
        "ZBC_13062016_filtered_chunk2.ldb"
    ],
    "screeningSettings": {
        "allowOmitFeatures": 0,
        "conformationMatchMode": "FIRST",
        "disableExclusionVolumes": false,
        "enableFragmentScreeningMode": false,
        "minRequiredFeatures": 3,
        "scoringFunction": "absolute",
        "screeningDatabasePropertiesDTOs": null,
        "timeOut": -1
    },
    "sgeExecutions": [
        {
            "sgeId": 19509,
            "sgeStatus": "Running"
        },
        {
            "sgeId": 19510,
            "sgeStatus": "Queued and Active"
        }
    ],
    "totalAmountCompounds": 192158
}
```

An important additional aspect of the web service API is extensibility. Future job types, such as conformer generation, can re-use the existing infrastructure to a large extent. This includes the error handling concept, the logging facilities, and response generation. The latter is done in an automatic way by converting Java DTO objects to their JSON representation by use of the Spring Framework's HTTP message converter support. The DTO objects are thereby provided as external library in order to enable clients to perform the same automatic conversion.

In order to deal with both invalid user inputs and internal server errors, a simple but efficient global exception handler is implemented using annotations provided by the Spring framework. The handler acts whenever a `RestController` throws an exception related to invalid user input, a custom `ApiException` or any other Exception that is not caught, and then returns an error message to the client. This concept makes sure that all otherwise uncaught exceptions result in error messages but also allows for fine-grained control over the handling of different exception types.

5.2 Job-Splitting Algorithm

Splitting virtual screening experiments into smaller sub-jobs and hence optimizing cluster resource utilization, as described in Section 4.3, is an important part of this work. The goal is to utilize all nodes of an HPC cluster, which requires special consideration since clusters are distributed-memory computers as introduced in Section 2.1.2. For this purpose, screening jobs are split into multiple sub-jobs, each running on one cluster node. The LigandScout VS command-line tool offers options to specify the first and last indexes of a range of compounds that should be screened. We use this functionality to create the desired sub-jobs. The amount of resulting sub-jobs is primarily dependent on the total number of compounds and the maximum number of compounds that are to be screened by a single sub-job. A default value for this parameter, already introduced as z, is defined in the server application's settings file (cf. Table 5.1), but can be overridden by users on a per-request basis for individual screening experiments. Concretely, each database is split into chunks of size z. In case the amount of compounds present in a specific database is not dividable by z, the remaining molecules are screened in an additional sub-job. The total number of sub-jobs is therefore

$$J = \sum_{i=1}^{n} \left\lceil \frac{c_i}{z} \right\rceil \ . \tag{5.1}$$

As introduced in Section 2.5.3, $\{c_1, c_2, ..., c_n\}$ denotes the set of compound database files. Algorithm 5.1 is applied in order to construct the respective `jobTemplate` objects that can be submitted to SGE via DRMAA. The `libSize` operation that is called during this algorithm is implemented by executing a LigandScout command-line tool. This program simply takes a database path as input and outputs the size of the contained compound library. Naturally, the given algorithm is only an abstraction of the concrete Java implementation. In particular, an actual `createJob` method used to construct

the jobTemplate objects, would also require additional parameters, such as the screening settings to be used.

Algorithm 5.1: Splitting a screening job into sub-jobs.

```
   Result: list of sub-jobs that can be submitted to the DRMS
 1 z ← maximum number of compounds to be screened per sub-job
 2 {f_1, f_2, ..., f_n} ← set of compound databases
 3 subJobs ← ∅
 4 for i ← 1 to n do
 5 |   c_i ← libSize(f_i)
 6 |   currIndex ← 0
 7 |   if z ≤ 0 then
 8 |   |   subJobs.add( createJob(f_i, currIndex, c_i − 1 ))
 9 |   else
10 |   |   while currIndex < c_i do
11 |   |   |   lastIndex ← currIndex + z − 1
12 |   |   |   if lastIndex ≥ c_i then
13 |   |   |   |   lastIndex ← c_i − 1
14 |   |   |   end
15 |   |   |   subJobs.add( createJob(f_i, currIndex, lastIndex ))
16 |   |   |   currIndex ← lastIndex + 1
17 |   |   end
18 |   end
19 end
20 return subJobs
```

Since all sub-jobs run completely independent from each other, they all produce separate output files. These have to be aggregated to deliver overall status updates to clients. Furthermore, management operations, such as cancelling of a job, require individual treatment of every sub-job related to the particular virtual screening experiment. Therefore, the server application uses the relational database to store meta-data for all sub-jobs. Figure 5.2 shows the fields of the resulting database table `screening_grid_execution`. These entities represent sub-jobs as they have been defined here.

Additionally to controlling the sub-job size, and thereby indirectly also the amount of sub-jobs to be started, users and administrators can adjust the slot consumption parameter u. As already introduced in Section 4.3, it specifies how many resource manager slots each sub-job should make use of. In the context of this thesis, resource manager slots always imply CPU cores. The key `sge.amount.slots` in the server application's settings file (cf. Table 5.1) is used to set the default value for u. Similar to z, it can be overridden on a per-request basis. Ideally, z and u are chosen so that enough sub-jobs of sufficient size are created to exploit all slots of the HPC cluster in use. An in-depth look on the possibilities with a single job on a cluster of four nodes is given in Figure 5.6. It shows how various

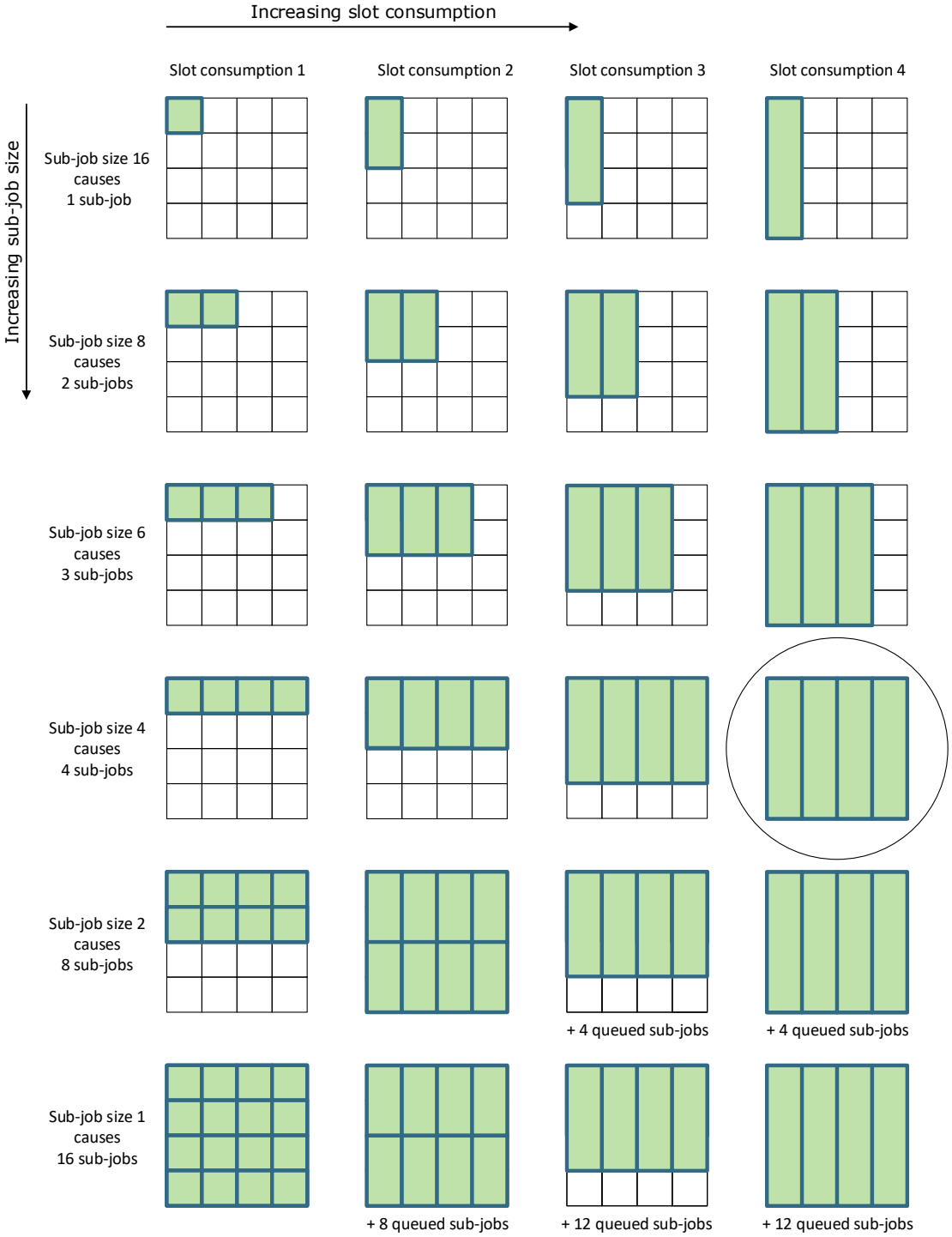

Figure 5.6: Effects of different job-splitting parameter values. A hypothetical VS job with an input database containing 16 compounds is split using various parameter combinations. The HPC cluster in use consists of four nodes, each of them contributes four slots. A column represents one cluster node, while a square within a column stands for a single slot that is provided by the respective node.

parameter combinations lead to different amounts of sub-jobs and also illustrates how the slot consumption parameter influences the distribution of sub-jobs across the cluster nodes. Under the assumption that the work done in each sub-job is homogeneous, the circle marks the most efficient parameter combination, which is to use all cluster resources, while starting as few sub-jobs as possible. However, starting more smaller sub-jobs might be beneficial in specific environments, where users are limited in the amount of slots their jobs can occupy at once. Chapter 6 provides detailed experimental results regarding the scalability of the implemented solution and the impact of the introduced overhead. We also analyze how the described parameters affect each other.

5.3 GUI Integration

One of the most fundamental questions of this thesis resolves around how to hide communication with the developed server application from the users of the client application and thus integrate transparent access to remote HPC resources into graphical desktop applications, in particular, the LigandScout molecular modeling and drug discovery suite.

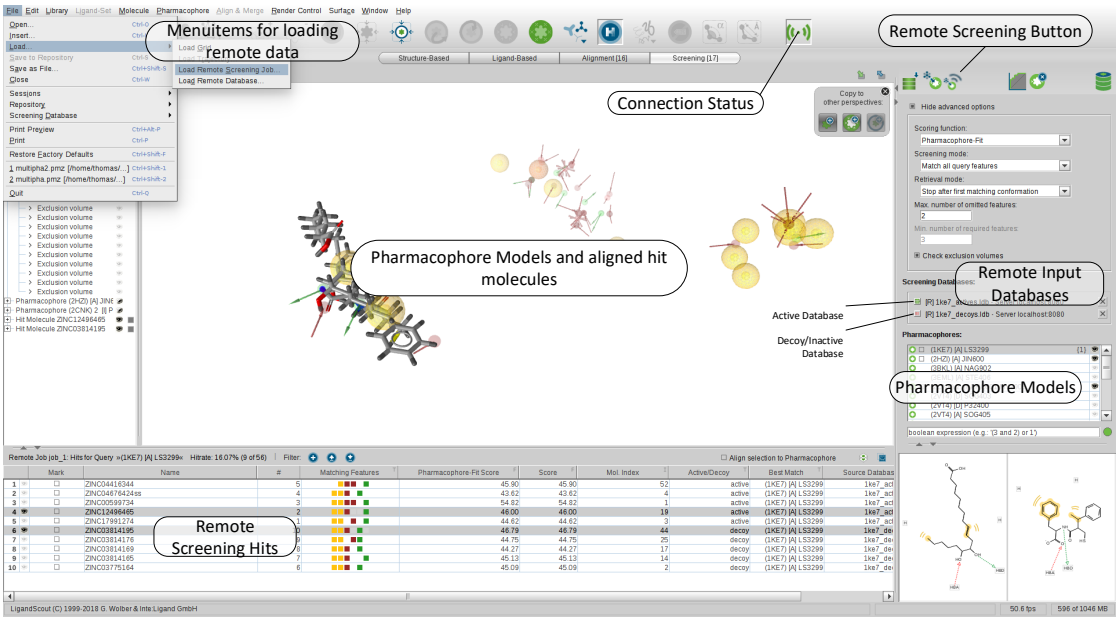

Figure 5.7: LigandScout screening perspective with new remote execution features.

The underlying client-side communication logic for this task is relatively straightforward. It is implemented using the Jersey Java Client [91]. Jersey is developed by Oracle and can be seen as the de-facto standard for developing RESTful Web Services. Concretely, we use the framework to construct all necessary client requests, which include complex HTTP multipart message types. Such requests are comprised of multiple different sets of data and are used to upload files along with associated metadata. Furthermore, the DTO classes used by the server application are provided as an external library. This

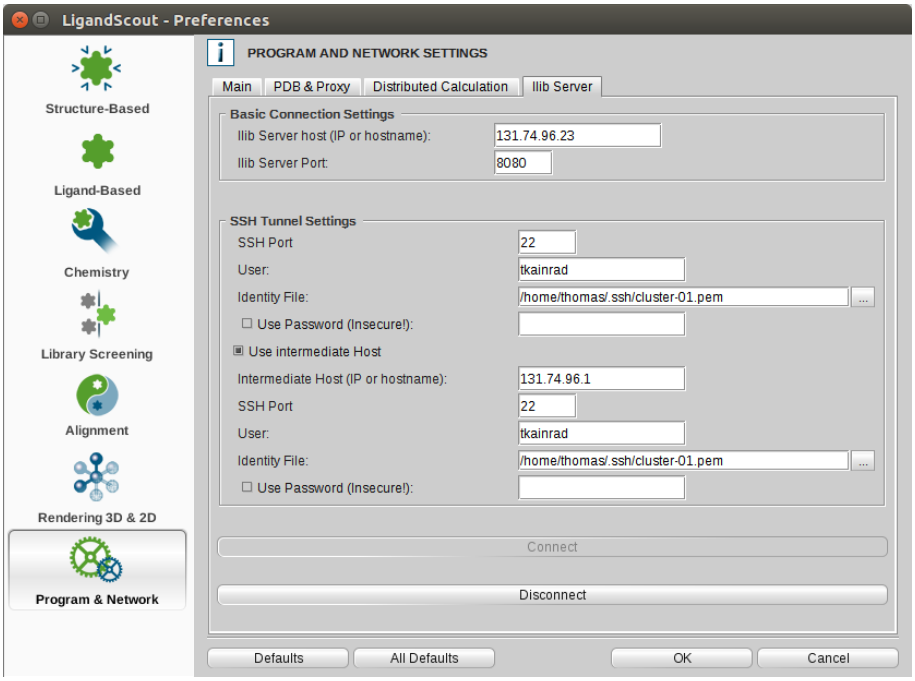

Figure 5.8: Configuration dialog for establishing SSH tunnels.

library is referenced by the LigandScout project and used to automatically generate JSON messages from the contained class definitions, similar to what is done on the server-side through Spring. This way, minor changes to the public web service API do not have to be manually considered within the GUI integration, as long as the API library is updated. The actual web service invocations are implemented by a single class. Other parts of the program can simply execute the respective Java methods for submitting screening jobs, gathering status information or downloading the results of a specific job. The newly implemented class, which adheres to the singleton pattern, also handles the creation of DTO objects from LigandScout internal data types.

More interesting is the graphical representation of the remote execution features. Figure 5.7 shows the LigandScout screening perspective after the applied changes. The already introduced Figure 2.11 can serve for comparison purposes. The most obvious addition is an icon in the main toolbar used for connecting to the server application. In this context, connecting means establishing the necessary SSH tunnels. The underlying logic regarding SSH connection management is implemented by use of Java Secure Channel (JSch), which is the de-facto standard SSH2 Java library. As already explained in the previous sections, the web service communication itself is completely stateless and does not depend on lasting session information. The current connection status is indicated by the color and state of the icon. Pressing the button, while a connection is already established, disconnects the server by destroying the current SSH tunnels. Naturally, users have to provide credentials before a secured SSH tunnel can be built. This is done

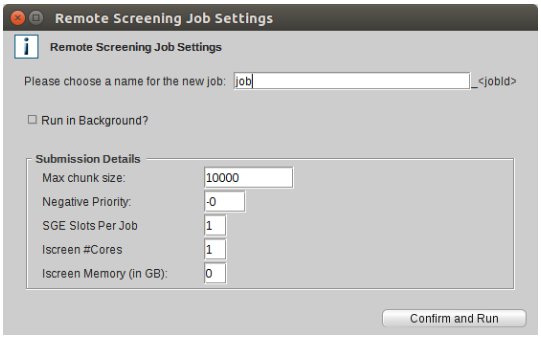

Figure 5.9: Submission dialog for new remote screening jobs. Before screening jobs are submitted to the remote cluster, users can name the job and override settings relevant for execution via resource manager software.

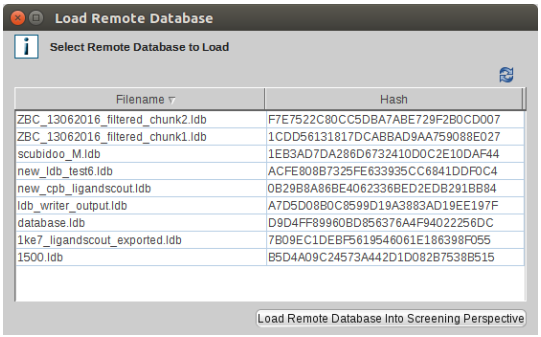

Figure 5.10: Dialog for selecting databases that are not available locally. If a database is not available locally, and therefore calculating the needed hash is not possible, this simple dialog can be used to choose compound libraries that should be screened remotely.

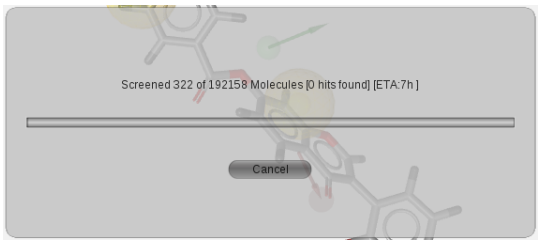

Figure 5.11: Progress information panel for regular local screening. Users have to wait for the screening to finish before they can continue working with the LigandScout screening perspective.

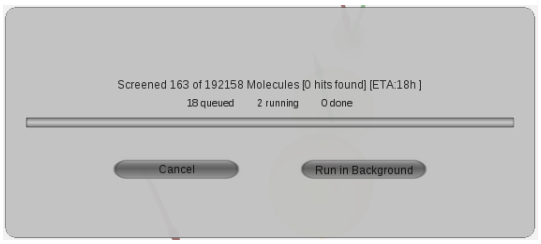

Figure 5.12: Progress information panel for remote screening. The panel can be hidden by pressing the *Run in Background* button, so that the LigandScout GUI can be used for other work while the screening experiment finishes on the remote cluster.

through the settings dialog displayed in Figure 5.8. Once the correct configuration is set, toggling between the connected and disconnected states is easily possible by single button clicks.

Starting a remote screening execution is done through a new button located right next to the existing icon for conventional local screening. The only additional dialog, that is shown during the submission process of a remote screening execution, is displayed in Figure 5.9. However, appropriate default settings are dynamically retrieved via a separate `GET` request from the connected server and should be viable in almost all cases. The average job submission therefore only requires one further click on the confirm button. The handling of the input data is completely transparent. Moreover, because of

the hashing approach described in Section 5.1.2, the regular LigandScout file choosing dialogues can be used for selecting compound libraries. Only when an input database is not available on the local storage volumes, a simple additional dialog for selecting remote databases is needed. Figure 5.10 depicts this dialog. It supports multi-selection and also offers a button for forcing a manual server-side refresh of the database list.

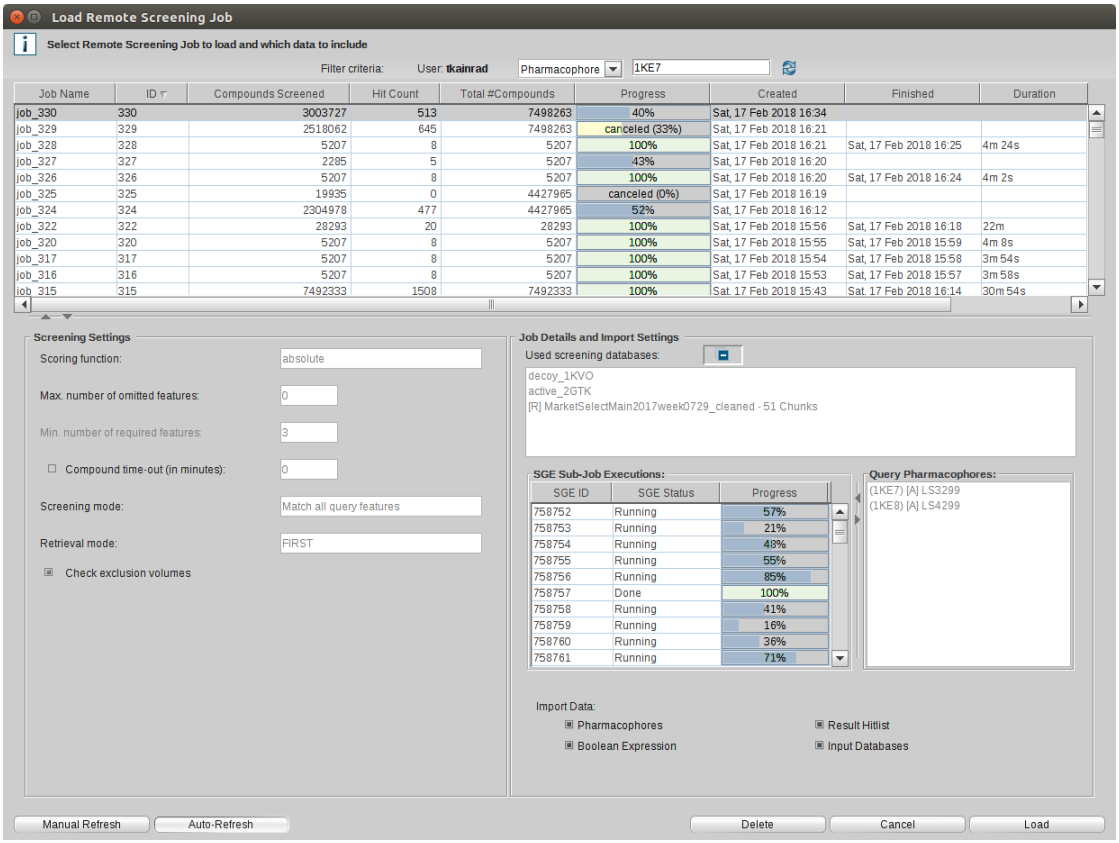

Figure 5.13: Dialog for monitoring and loading remote screening jobs.

By default, the remote screening process provides progress updates in the foreground of the graphical interface, almost identical to a local screening run. Figures 5.11 and 5.12 serve as a comparison between the two. However, apart from the obvious benefits, such as shorter execution times and decreased load on the local machine, the developed remote execution solution offers several usability advantages over local screening. Most importantly, the progress dialog can be hidden, enabling users to continue other work within the LigandScout GUI. This can include preparing and starting of additional remote screening jobs. In order to monitor and manage all past jobs, which have been sent to the currently connected server, an additional dialog is implemented. It is shown in Figure 5.13 and provides a comprehensive table of all remote screening runs, gives detailed information about their status, and displays the settings that were used to start a specific job. Per default, the table shows all jobs started by the current user, but allows

filtering by pharmacophore or database name. The displayed progress information is automatically updated, resulting in an animated progress bar for each table row. A convenient implication of this concept is that various screening experiments, using the same input data but slightly different settings, can be easily compared. Furthermore, the dialog is the starting point for downloading input data as well as (intermediate) results into the LigandScout GUI, where they can directly be used for further experimentation. This download process is feasible because the amount of result data that has to be transferred is kept to a minimum. For any reasonable virtual screening experiment, the amount of hit compounds will not lead to immense data sizes, otherwise the pharmacophores have not been specific enough. The dialog further enables users to choose which kind of data should be downloaded. Moreover, jobs can be canceled and deleted. The latter also frees storage by erasing all server-side files related to the particular virtual screening experiment.

5.4 Cloud Deployment

The design decisions described in Section 4.5 are made to guarantee that the server application works well within cloud environments. To capitalize on this benefit, an automatic way for deploying HPC cloud clusters, which are ready to use for virtual screening, is needed. Amazon's EC2 Cloud and the CfnCluster toolkit provide the required technology to accomplish this.

CfnCluster offers a large amount of possible configuration parameters. These range from straightforward settings, such as the operating system to be used for the created cluster nodes, to network-related settings. In our configuration, we use Ubuntu 16.04 as the base operating system and SGE as the DRMS. Furthermore, an Elastic Block Storage (EBS) snapshot is created that contains a pre-configured server application installation as well as several compound databases. This snapshot is specified in the CfnCluster configuration to serve as the template for the shared storage volume of all cluster nodes. All nodes are further created in a designated Virtual Private Cloud (VPC). In real-world scenarios, the usage of VPCs has indispensable security benefits. A VPC is a logically isolated network that can be configured freely. Combined with virtual private network (VPN) technology, VPCs can be incorporated into an organization's local network in order to exploit both the benefits of cloud computing and a coherent organization network. Another important configuration option is the type of compute instance to use. Along with the number of used nodes, this is the main cost factor. Amazon offers a large variety of different specifications, starting with one virtual CPU per instance and ranging up to 64 cores per virtual machine. Of course, the developed server application is intended to work with compute instances of all sizes. Section 2.5 explores both horizontal and vertical scalability.

The second part of necessary prerequisites for a CfnCluster deployment is the *post install script*. The thereby defined actions are executed on every cluster node after the automatic base setup is finished. This initial base setup includes configuring the network, connecting

the shared storage volumes, and initializing the resource manager and job scheduler. In order to be able to submit screening jobs immediately after cluster creation, without any manual configuration efforts, it is required that the specified script activates the LigandScout license for the pre-installed version present on the shared EBS volume. Apart from this, the most important step is, of course, to start the execution of the server application. Unfortunately, the SGE version that is used by cloud clusters created with CfnCluster is compiled without Java DRMAA support. Therefore, the post install script also moves pre-compiled native libraries to the appropriate places within the SGE installation. Appendix E provides an exemplary configuration file and the used post install script.

Having the configuration file and post install script in place, the commands shown in Listing 5.2 can be used to create, monitor, stop, and delete an HPC cloud cluster. Figure 5.14 illustrates the resulting deployment architecture.

Listing 5.2: Most important CfnCluster commands.

```
1 cfncluster create hpc-cluster;
2 cfncluster status hpc-cluster;
3 cfncluster stop hpc-cluster;
4 cfncluster delete hpc-cluster;
```

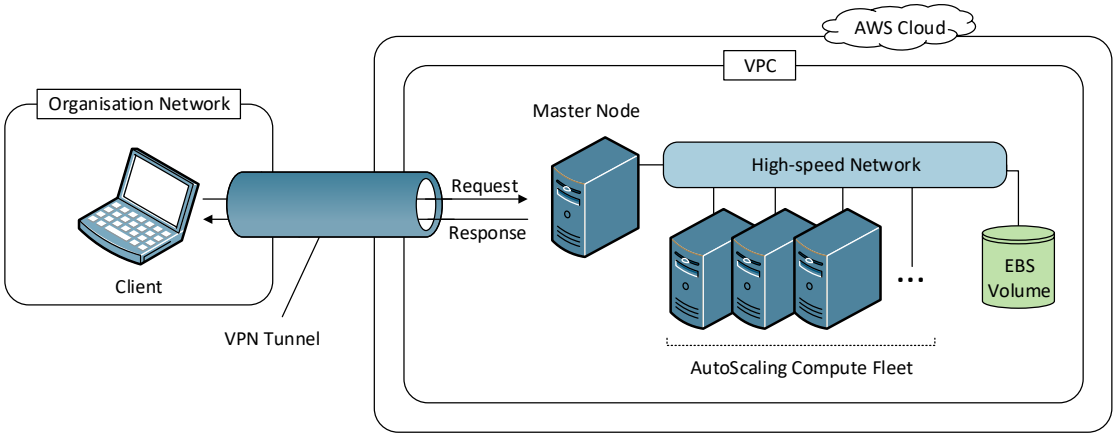

Figure 5.14: Possible AWS deployment setup. By use of VPN technology, an organisation's network can be connected with an AWS VPC in order to establish a single shielded network. This has mainly practical benefits, secure communication is already guaranteed by connecting via SSH tunnels.

CHAPTER 6

Results and Discussion

This chapter examines the performance and scalability of the developed solution by presenting the results of various benchmarking experiments. Furthermore, we present an analysis on the cost of HPC clusters in the cloud. The chapter concludes with a discussion about whether the implemented GUI integration comes with the envisioned usability benefits.

6.1 Performance Evaluation

The primary goal of our evaluation is to assess the performance improvements that can be accomplished by using the developed remote execution solution instead of relying on local consumer hardware. Furthermore, the scalability of the job-splitting approach presented in Sections 4.3 and 5.2 is investigated. An additional focus lies on the performance that can be obtained by HPC clusters in the AWS EC2 cloud. For this purpose, clusters with different node counts and varying underlying virtual machine instances are created.

6.1.1 Benchmarks

The amount of time spent to compute $s(m_i, d_j)$, which decides whether d_j fits pharmacophore model m_i, depends both on the model m_i and the compound in question d_j. Specifically, it is relevant how complex both components are and whether the compound fits the model or can be discarded already after the first distance checks. Therefore, the models and molecule libraries used for performance evaluation affect the total runtime. As a consequence, absolute values of experiments with different input data, even if the number of molecules and models is equal, can not be compared. However, as long as the same input data is used for all experiments, the characteristics of the data are of little relevance for our performance evaluation.

The main benchmark used in the following sections relies on a chunk of the ZINC molecule database [92] and a classical LigandScout pharmacophore model for the cyclin-dependent kinase 2 (CDK2) target. The corresponding enzyme plays an important role in cancer research [93, 94]. The used database chunk contains 149 187 compounds and represents some of the most commonly used molecules for virtual screening. This is because the ZINC database is freely available even though it contains well annotated and commercially available compounds. As one of the main goals is to compare the performance with consumer hardware, the database size for this benchmark has been chosen so that virtual screening experiments are still feasible also on this type of machines. The exact number 149 187 simply stems from the amount of molecules present in one of the smaller ZINC database chunks used at the Department of Pharmaceutical Chemistry at the University of Vienna. The whole ZINC database as published by Irwin et al. [92] contains 727 842 molecules and is too large for the planned benchmarking experiments.

In order to exhaust the capabilities of the most powerful HPC cluster that has been available for this work, we perform an experiment with the same input database, but an increased amount of query pharmacophores. The resulting experiment can also serve as an example to illustrate the applicability of the developed software to activity profiling workflows.

6.1.2 Environment and Methodology

Four different systems are used for running the benchmarks. These are the following:

- **`notebook`** Lenovo Thinkpad T440s consumer notebook. It contains an Intel Core i7-4600U CPU with two physical cores working at a clock speed of 2.1 GHz. Intel's hyper-threading technology is supported. The notebook can further resort to 12 GB of RAM. Ubuntu 16.04 LTS is used as the operating system.
- **`workstation`** Desktop PC containing an Intel (R) Core(TM) i7-3770 CPU with four physical cores running at a base clock speed of 3.40 GHz. The system supports hyper-threading and has 32 GB of main memory. The operating system used for running the tests is CentOS 7.4.
- **`hydra-cluster`** Cluster located at the University of Vienna comprised of 15 machines. Each node comes with an Intel(R) Xeon(R) e5-2630 v3 CPU, which operates 8 cores at a clock speed of 2.40 GHz. Hyper-threading is supported, which means that each node provides 16 slots. The individual cluster nodes further possess 64 GB of RAM. CentOS 7.4 is used as the operating system.
- **`aws-cluster`** HPC cluster created dynamically by use of the Amazon EC2 cloud and the CfnCluster toolkit. The operating system installed on all instances is Ubuntu 16.04. The number of nodes used for a specific experiment and the exact specification of the compute nodes are given with each individual benchmark run.

All applications, which includes the server program, the LigandScout command-line tools, and the LigandScout GUI application, are executed by use of the Oracle JDK 8u151. Unless specified otherwise, all values are measured three times. The standard deviations of the different runs are plotted as error bars. The experiments that are done in the AWS cloud are performed only once. This is because those experiments are costly and have a longer runtime, which leads to more stable results and circumvents the need for multiple executions. The measurements are taken client-side and the clock is started before the first request, which means that any network communication is already included in the observed runtime values. Upon receiving a screening request, the server has to apply the job-splitting algorithm to the requested databases and submit all resulting sub-jobs to the resource manager. Consequently, the time spent for these tasks is also included.

In order to evaluate our job-splitting algorithm and compare different parameter combinations, several experiments are performed for different sub-job sizes and therefore also for different amounts of sub-jobs. Please refer to Sections 4.3 and 5.2 for detailed explanations on these concepts. Figure 5.6 further clarifies how different job-splitting parameters influence the virtual screening execution on distributed-memory clusters. However, in all cases, the job-splitting parameters are chosen so that the available cluster nodes are fully used, which means that no slots remain idle. As a consequence, in our experiments, a lower sub-job slot consumption goes hand in hand with an increased amount of sub-jobs, because the sub-job size has to be set lower, which causes more and smaller sub-jobs.

6.1.3 Results

In the following, we present the results of our performance evaluation. The performed experiments are grouped by the type of machine that has been used for their execution.

Local Screening

The conventional way of performing virtual screening experiments through the LigandScout GUI relies on the computational power of a user's personal notebook or workstation computer. The runtime and throughput values that can be obtained on this type of hardware are visualized in Figures 6.1 and 6.2. These results are a good illustration of the problem, that this work aims to solve. Even though the used compound library contains a relatively small number of compounds, and the pharmacophore-based virtual screening process is considered very fast, the experiment takes more than 50 minutes on the `notebook` and still almost 20 minutes on the `workstation` computer. Drawing a concrete comparison between the `notebook` and the `workstation` is of little relevance for our evaluation purposes. Nevertheless, it can be noted that the performance difference is roughly what can be expected due to the difference in hardware. The theoretically expected speedup can be roughly approximated as

$$\frac{4 \times 2.1\,\mathrm{GHz}}{2 \times 2.1\,\mathrm{GHz}} \approx 3.24 \quad . \tag{6.1}$$

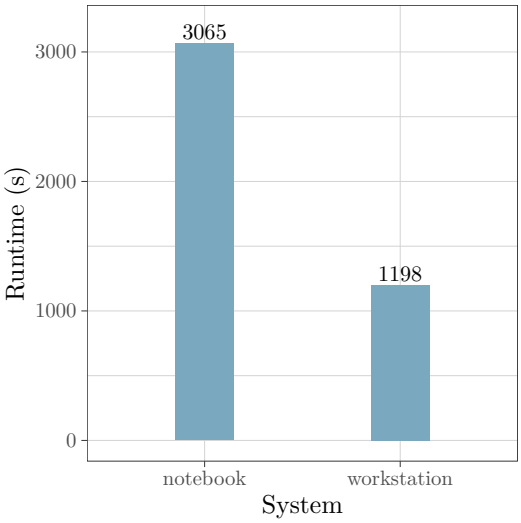

Figure 6.1: Runtime on `notebook` and `workstation`.

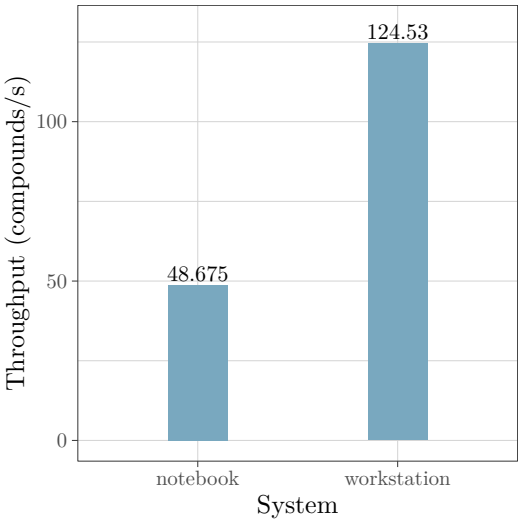

Figure 6.2: Throughput on `notebook` and `workstation`.

The observed speedup is

$$\frac{3065\,\mathrm{s}}{1198\,\mathrm{s}} \approx 2.56 \quad , \tag{6.2}$$

which almost matches our expectations. The difference between expected and observed speedup stems from the fact that we only took the compute power into consideration, but screening jobs are mostly memory (I/O)-bound.

The throughput metric that is used in Figure 6.2 refers to the amount of compounds that are screened on average per second, hence, it is inversely proportional to a job's runtime. Scientists in the field of molecular modeling are dependent on virtual screening for validating their models. Typically, a screening run against a reference molecule library is done after every model refinement step. Furthermore, the virtual screening process is dependent on various parameters, such as the number of pharmacophoric features that have to match. These parameters can be changed by the user, resulting in a demand to perform multiple screening experiments, each using different parameter values. In short, waiting 50 minutes or even 20 for the completion of a small virtual screening experiment is a bottleneck in the early stages of the drug discovery workflow.

Larger jobs using millions of compounds and more demanding parameter settings are barely feasible to run on local machines at all.

On-site HPC cluster

The `hydra` cluster of the Department of Pharmaceutical Chemistry at the University of Vienna is an excellent system for testing the performance of the developed solution. A starting point is given in Figures 6.3 and 6.4, which show the runtimes and throughputs

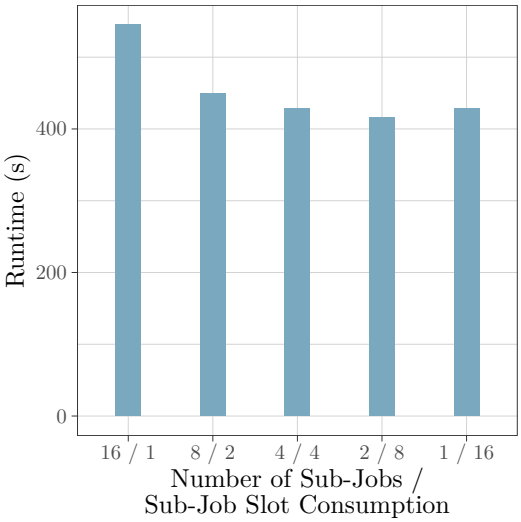

Figure 6.3: Runtime using a single node of `hydra`.

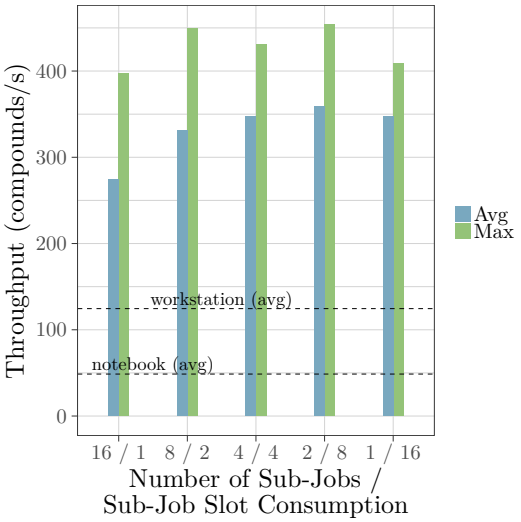

Figure 6.4: Throughput using a single node of `hydra`.

that can be obtained using a single node of the cluster. The values are given for different numbers of resource manager jobs that are started in parallel. However, in all cases the single cluster node is fully used. A single node of the `hydra` cluster provides 16 slots, one for each core plus eight additional slots through hyperthreading. Using the developed server application, it is therefore possible to start either a single screening job using all 16 available slots or to split the virtual screening experiment into several sub-jobs. The latter has significant practical advantages. Most importantly, it is not necessary to wait until all slots of a single node are free, instead the sub-jobs can be distributed over several nodes of the cluster. Additionally, some sub-jobs can be started immediately while others wait in the queue until enough slots are available. The results obtained by using just one cluster node show that this job-splitting approach comes with little overhead. Only when 16 sub-jobs are started on a single node, the performance drop is considerable. This is expected, because the nodes only comprise eight physical cores and relying on hyper-threads for half of the started sub-jobs necessarily leads to performance degradation.

Another interesting detail can be derived by comparing the performance to local screening. The server hardware of the compute node performs almost three times faster than the workstation computer, even though it contains only twice as many CPU cores. This observation can serve as an additional argument for doing lengthy computational work on dedicated HPC hardware, which comes with many optimizations for computationally expensive tasks.

The most important results regarding horizontal scalability on `hydra` are visualized in Figures 6.5 and 6.6. They show the strong scaling capabilities of the proposed solution

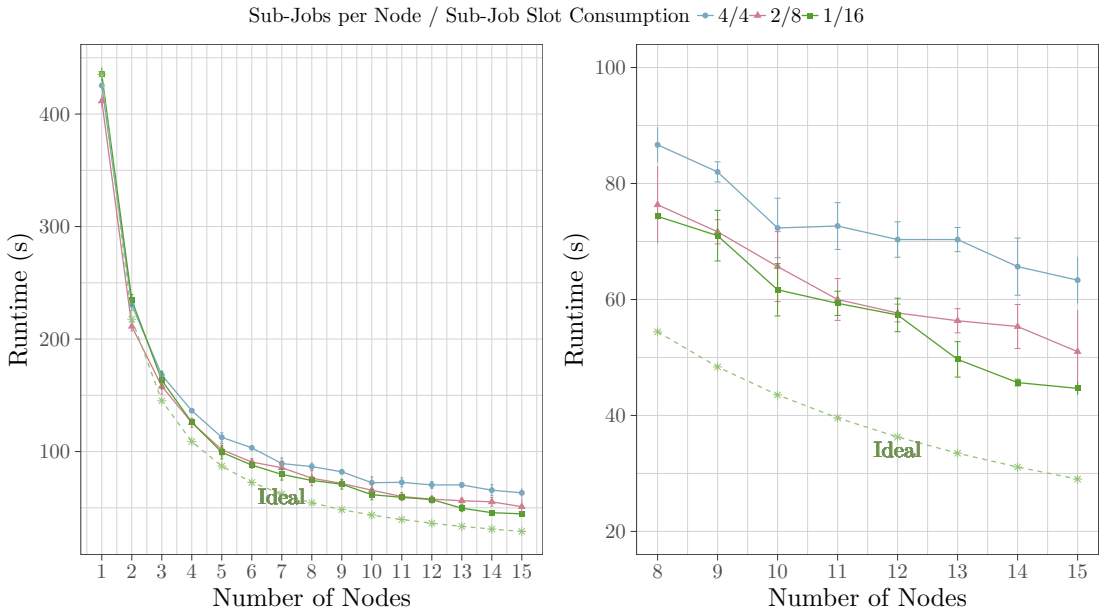

Figure 6.5: Strong scaling analysis on `hydra` using different sub-job sizes.

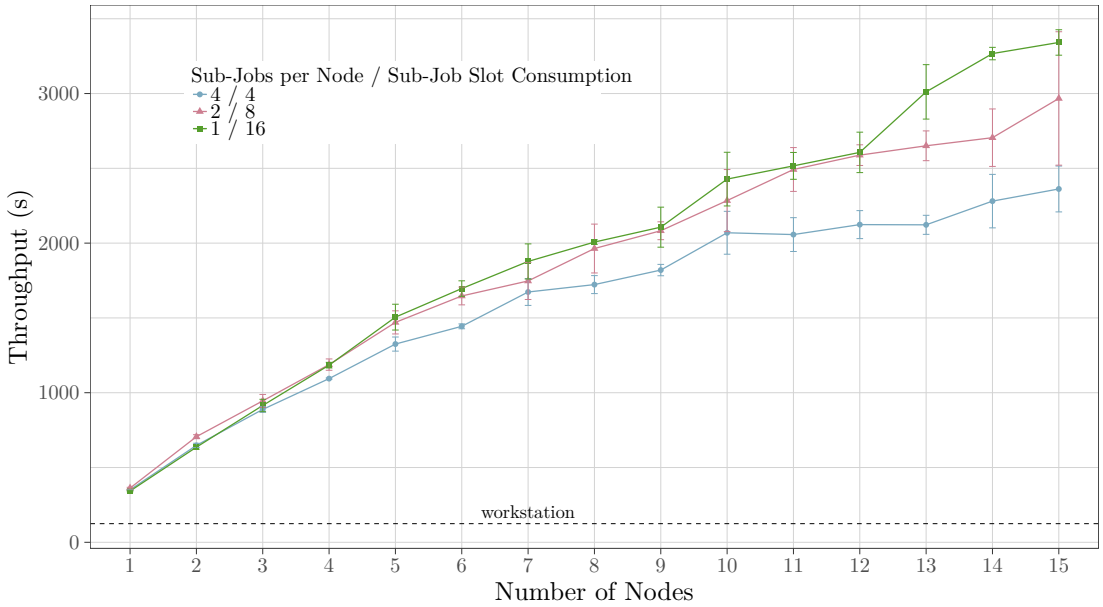

Figure 6.6: Strong scaling analysis on `hydra` using the throughput metric.

using different sub-job slot consumption parameter values. It is shown that the virtual screening experiment which takes close to 20 minutes on a recent workstation computer, can be done in 45 seconds on a HPC cluster of 15 nodes. Once again, it is important

to note that a lower slot consumption value u means that more sub-jobs are started in parallel. As already explained, this parameter refers to the amount of slots that are occupied by each started sub-job. For these scaling experiments, the sub-job size z has been chosen dependent on u and the number of nodes in use. The goal has been to create exactly as many sub-jobs as are needed to fully exploit all available resources. This ideal amount of sub-jobs is defined as

$$J^{ideal} = \frac{|Nodes| * 16}{u} \ . \tag{6.3}$$

Using Equation 5.1, we can calculate the sub-job size to use by

$$z^{ideal} = \left\lceil \frac{149187}{J^{ideal}} \right\rceil \ . \tag{6.4}$$

Users of the LigandScout GUI should not have to deal with these parameters. However, the evaluation experiments show, that the sub-job size does affect the scaling efficiency. Administrators are therefore advised to find reasonable default settings for their clusters. In particular, if the amount of compounds to be screened by each sub-job becomes too low, the increased overhead begins to neutralize added computational resources. Considering that a single `hydra` node is capable of screening more than 400 molecules per second, it is obvious that the overhead of submitting the job to the resource manager and the initialization tasks performed by the LigandScout virtual screening tool can become overwhelming for very small jobs. Figure 6.7 provides another view onto this issue. It shows how the runtime increases when the input data is split into smaller chunks and therefore more sub-jobs are started. There is once again a positive correlation between the number of sub-jobs and the runtime. However, it is also clear that the scaling limitations are not primarily introduced by the job-splitting algorithm, but rather correlate strongly to the amount of molecules to be screened during each sub-job.

In order to assess the scalability for larger screening experiments, we also screened the same database using multiple input pharmacophores, a technique already introduced as activity profiling. For this purpose, a diverse set of 25 models for targets from the DUD-E [95] database has been built. They have been randomly selected, because the actual targets and models are of little importance for this analysis, instead the goal is simply to lengthen the overall runtime and analyze the effects on scalability. Figure 6.8 shows the results of this additional strong scaling analysis with a diverse set of 25 pharmacophore models, but the same database of 149 187 compounds. Because of substantially longer runtimes, the results of this particular benchmark have been measured only once. The scaling performance in this experiment is close to ideal, as the fraction of time spent for initialization and the job-splitting algorithm is lower. Actual real-world screening experiments will often be even larger. In conclusion, it can be noted that each sub-job should have at least enough input assigned to run close to one minute. Otherwise, the overhead of the network communication and the job-splitting and submission process consumes a significant portion of the total runtime. However, as actual virtual screening experiments often have to deal with millions of compounds, this is not a problem in

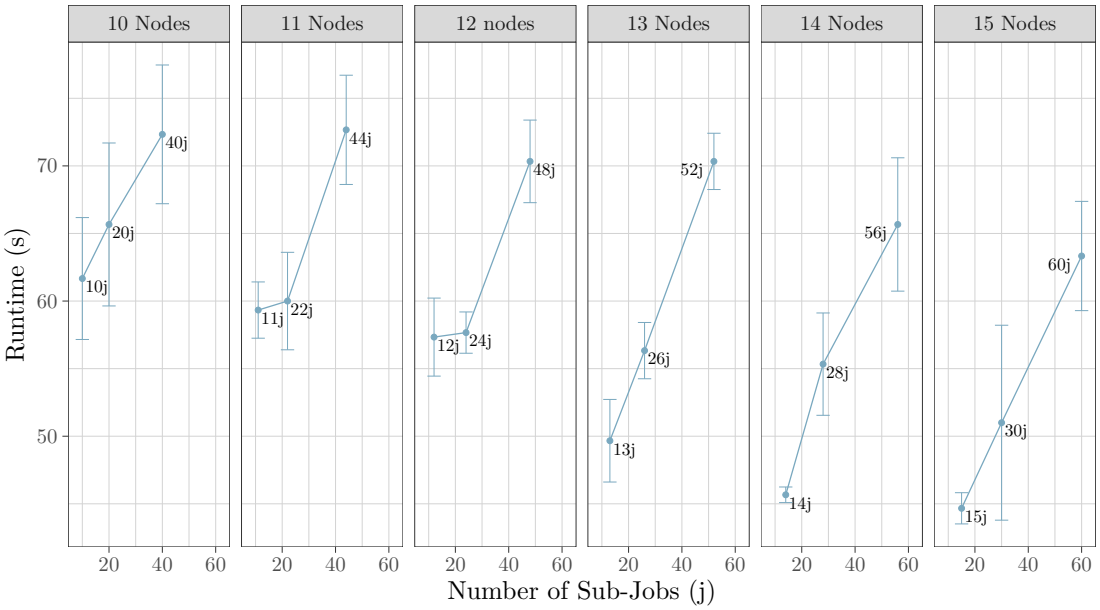

Figure 6.7: Impact of increasing number of sub-jobs on runtime. The lanes represent the number of cluster nodes that are used while the three points within each individual lane correspond to three different sub-job slot consumption values, namely 16, eight, and four. Because the jobs are split so that all can be started at once and no slots remain idle, the sub-job sizes are adjusted to the slot consumption and lead to the depicted numbers of sub-jobs.

practice. Activity profiling, which comes with larger problem sizes due to an increased amount of query models used in parallel, is another use case, where initialization times of a few seconds are negligible. In practice, the almost constant initialization time that is needed for the initial remote submission process is therefore rarely an issue at all.

The presented observations underline the purpose of the various parameters that can be set in order to customize the job-splitting algorithm. There is a trade-off between using many small sub-jobs with low slot consumption as opposed to relying on as few sub-jobs as necessary in order to fully exploit the cluster. The former allows the job scheduler of the underlying DRMS to flexibly schedule the sub-jobs across several nodes. The latter has a slightly lower overhead, which results in theoretical performance benefits. On the other hand, sub-jobs with high slot consumption might have to wait longer in the queue of the job scheduler, because they demand more resources of an individual cluster node to be available at once. As a consequence, the performance gains can easily be negated by increased waiting times.

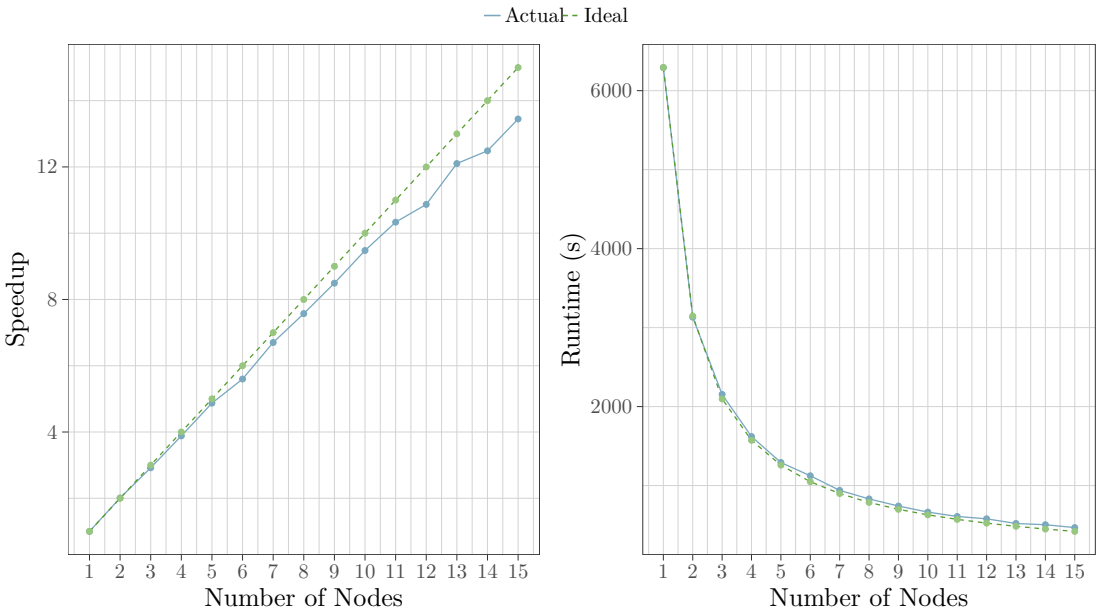

Figure 6.8: Strong scaling analysis on `hydra` using a larger job with 25 pharmacophores.

Amazon Web Services (AWS) EC2 cluster

In order to show the applicability of HPC in the cloud, we deploy different clusters in the AWS EC2 cloud. Table 6.1 lists the configurations of the specific instance families used in our experiments. The concrete instance types that are representatives of these families are listed in Table 6.2.

Table 6.1: AWS instance families used for benchmarking.

Instance Family	**Use Case**	**CPU Model**
m4	General Purpose	Intel Xeon® E5-2686 v4 (2.3 GHz) or E5-2676 v3 (2.4 GHz)
c4	Compute optimized	Intel Xeon® E5-2686 v4 (2.3 GHz) or E5-2676 v3 (2.4 GHz)

Even though it would be an interesting use case to create completely new cloud clusters for individual screening jobs, this is not done for the below experiments. New deployments take several minutes before they are fully ready for screening, depending on a multitude of parameters such as the amount of nodes, the used operating system, the type of machine instances and many more. This is not feasible for running relatively short benchmarks. Also in practice, we recommend to have a small, but fully configured cluster continuously running. The costs for running only a small master node are negligible

Table 6.2: AWS instance types used for benchmarking.

Instance Type	vCPUs	Memory (GiB)	Costs ($/h)
`m4.large`	2	8	0.12
`m4.xlarge`	4	16	0.2
`m4.2xlarge`	8	32	0.4
`m4.4xlarge`	16	64	0.8
`c4.large`	2	3.75	0.114
`c4.xlarge`	4	7.5	0.24
`c4.2xlarge`	8	15	0.48
`c4.4xlarge`	16	30	0.96

when compared to any kind of hardware investment. In all presented experiments, a `t2.micro` instance is used as master node, which does not run computations itself but hosts the server application. Running a `t2.micro` instance currently costs 0.01116$ per hour. Additional resources in the form of virtual machine instances can then be added to the cluster on demand in less than one minute, once again depending on a number of parameters. In the future, startup times for newly created clusters may decrease even further, which could enable fully dynamic deployments for individual small screening experiments.

The strong scaling results for clusters consisting of `m4.large` and `c4.large` instances are visualized in Figures 6.9 and 6.10, respectively. Essentially, the observations from the previous experiments using `hydra` are reinforced. The scaling efficiency is once again good, consistently better than a theoretical program which is 95% parallelizable. This is the case even for the relatively small screening experiment that was used for benchmarking. It is interesting to note, that the compute optimized `c4.large` instances do not only perform better in general, but also scale more efficiently. Apparently, the c4 instances are able to decrease the time needed for initialization purposes. The results further illustrate the need for evaluating different cloud computing offerings. The general purpose `m4.large` instances are more expensive than their c4 counterpart due to having more RAM available. Nevertheless, for the virtual screening use case the `c4` instance family proves to be more efficient. One `c4.large` virtual machine instance is already able to outperform the `notebook` machine, while a cluster consisting of three instances is significantly faster than the `workstation` machine. Figure 6.11 shows the throughput results obtained by using cloud clusters of different sizes. The newly introduced maximum throughput value denotes the highest average throughput achieved during any one-minute interval. It can be expected that this metric is more forgiving when it comes to initialization and management overhead. The chosen maximum throughput interval will likely be in the middle of a job's execution, when this kind of additional

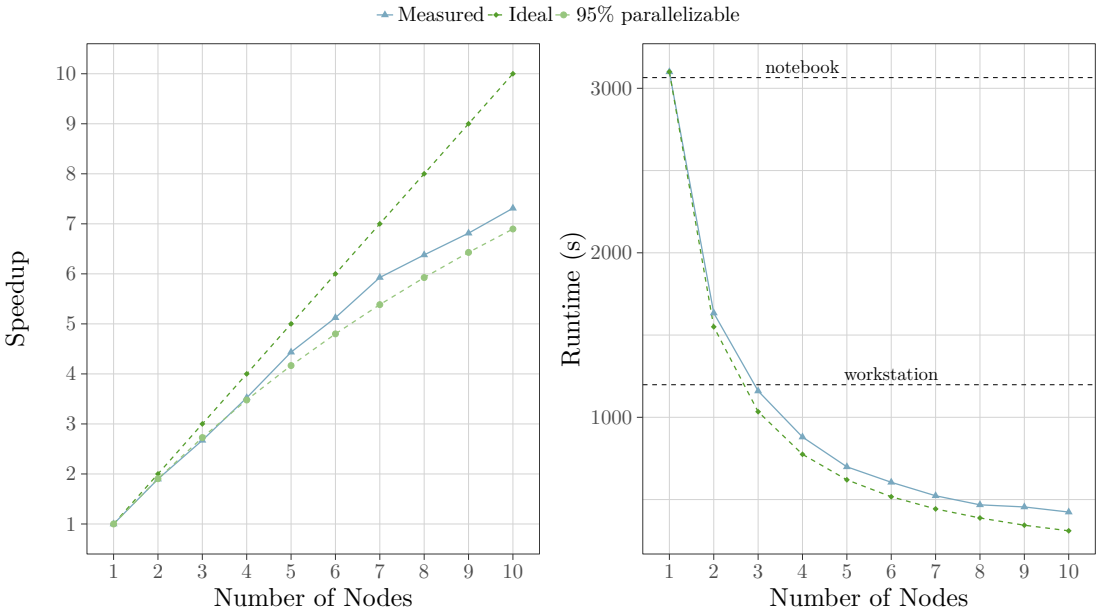

Figure 6.9: Strong scaling analysis using `m4.large` instances.

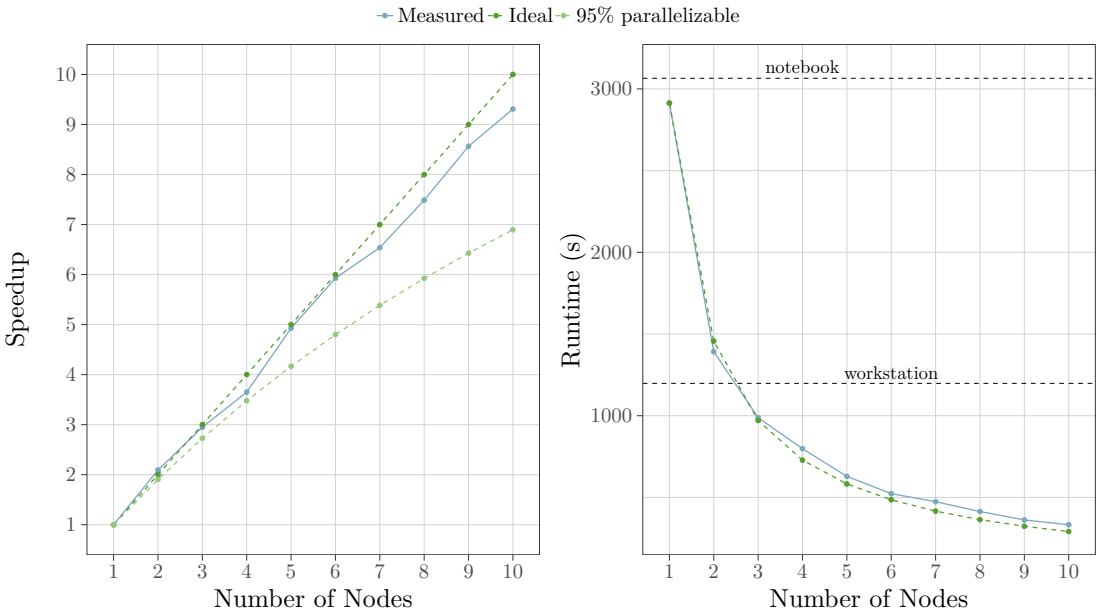

Figure 6.10: Strong scaling analysis using `c4.large` instances.

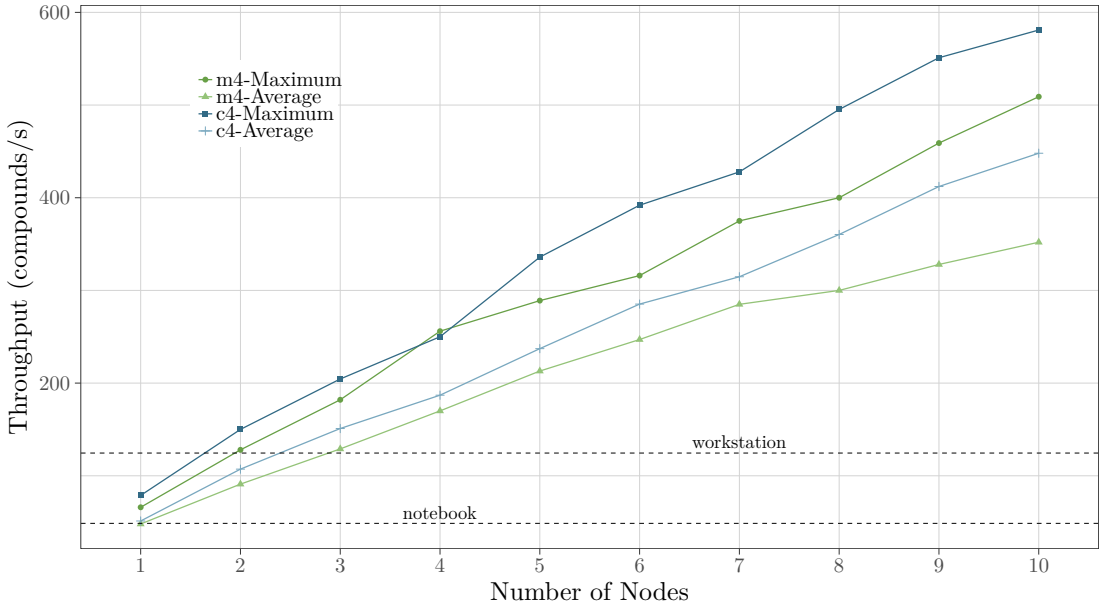

Figure 6.11: Strong scaling analysis in the EC2 cloud by use of throughput.

work is not necessary. Once again, it can be seen that a small cloud cluster of three `c4.large` instances is already able to outperform the `workstation` machine.

Additionally to analyzing the ability of scaling with a growing number of nodes, we evaluate our solution's vertical scaling capabilities, i.e. how it can scale with different instance types providing an increasing number of CPU cores. To this end, different AWS EC2 instance types are used with cloud clusters of 2 nodes. Once again, the benchmark runs are done multiple times in order to analyze the impact of different sub-job sizes. Figure 6.12 presents the results of these experiments. In all cases, the c4 instance family outperforms the m4 instance family. The vertical scaling efficiency is close to optimal, especially when larger instances are allowed to make use of larger sub-job sizes. It is interesting to note, that clusters consisting of two `m4.4xlarge` instances provide slightly better performance than two `hydra` nodes. Both node types offer 16 threads to the resource manager. While the newer and more powerful CPU models used by AWS EC2 can generally be expected to perform better, the cloud instances also have to deal with virtualization overhead. Our experiments suggest that AWS is able to keep this overhead minimal.

Figure 6.13 provides a novel view onto the performance degradation inflicted by using a large number of sub-jobs. In the previous experiments, the total amount of sub-jobs has always been chosen so that the cluster resources are sufficient to start all sub-jobs immediately. Naturally, submitting more jobs than the available resources can start at once leads to increased runtime due to initialization tasks at the beginning of each sub-job. Recall Equations 6.3 and 6.4 that have been used to calculate the ideal sub-job

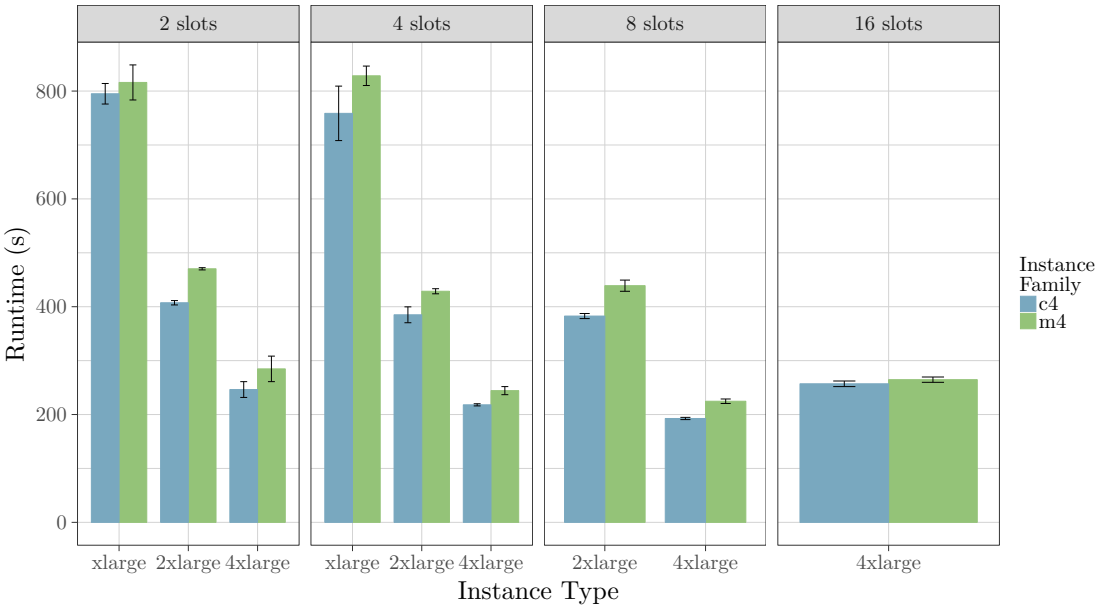

Figure 6.12: Vertical scaling analysis using cloud instances. Each of the plots corresponds to a different sub-job slot consumption value. When looking at the large instance types, a lower slot consumption means that more sub-jobs are started in parallel. The analysis focuses on the performance gains that can be achieved by using larger AWS EC2 virtual machine instances.

size. The goal of this experiment is to evaluate the dimension of this additional overhead. The data points on the left side show the average throughput values for all 1-minute intervals within a job's execution. Naturally, throughput is worse at the beginning, when a fraction of the time is needed for initialization purposes and network communication, as well as in the end when some sub-jobs are already finished, causing cluster resources to be unused. The most interesting comparison can be drawn from the middle intervals. Using 30 or even 50 sub-jobs on a cluster containing 10 virtual machine instances results in an expected throughput decrease. An increased amount of sub-jobs leads to smaller job sizes in regards to the amount of compounds to be screened during each individual sub-job. In this experiment, all sub-jobs independent of their size make use of two resource manager slots, which is equal to a full `m4.large` instance. As a result, more and smaller sub-jobs typically do not span the whole execution time of the screening experiment. Instead, sub-jobs are executed sequentially, causing additional initialization and management overhead. The density plot on the right side of Figure 6.13 shows that starting three times as many sub-jobs as can be run immediately leads to a 28% decrease in mean throughput, while using five times as many jobs results in a 44% decrease. While these numbers are certainly significant, specific organizational cluster usage policies may still reward high numbers of small sub-jobs. In general, it is beneficial to correlate the amount of sub-jobs and their total slot consumption with the size of the HPC cluster at hand.

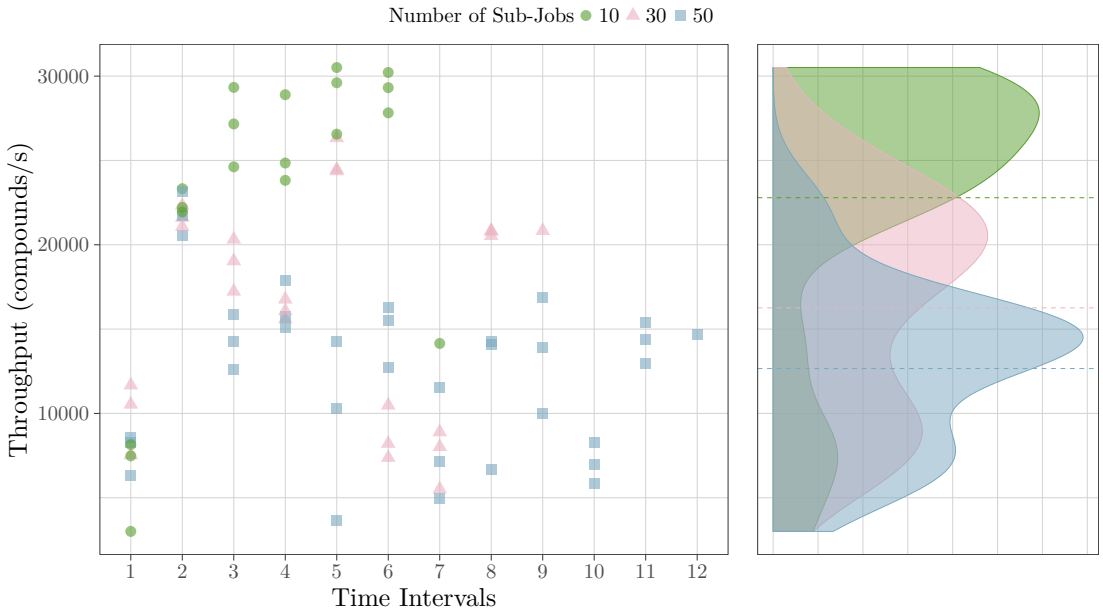

Figure 6.13: Impact of using many small sub-jobs versus few large ones. The scatter plot shows the throughput for all 1-minute intervals during a job's execution on a cluster in the AWS EC2 cloud using `m4.large` instances. Experiments have been performed three times, hence three values are plotted for each amount of sub-jobs and each time interval.

6.2 Discussion

This section summarizes the main results of the above presented experiments. Furthermore, additional aspects such as the cost of cloud clusters are highlighted. The section is concluded with an analysis regarding the usability of the LigandScout GUI integration.

6.2.1 Performance and Scalability

Benchmarking has shown that the developed job-splitting approach is indeed capable of exploiting large distributed-memory clusters. As long as the sub-job size is reasonably large, the non-parallelizable management and initialization overhead is almost negligible. In the context of this work, it is especially important to compare the performance of client machines, such as notebook and workstation computers, against the capabilities of HPC hardware. It has been shown that even small virtual screening experiments can easily take more than 50 minutes on portable notebooks and still 20 minutes on recent desktop workstations. These are already numbers that are difficult to work with when many screening runs have to be done in order to perform model refinement. For large-scale virtual screening experiments, using millions of input compounds with the goal of finding novel, early-stage drug candidates, this level of performance is not feasible.

Reducing the runtime from 20 minutes to less than 45 seconds is therefore an invaluable improvement.

The performed evaluation experiments further illustrate the flexibility of the developed solution. It can scale well, both horizontally and vertically, by providing various parameters to cluster administrators and also to advanced users on a per-request basis. These parameters also enable the server application to be deployed on clusters with strict usage policies. For example, it might be required that resource manager jobs occupy only a fixed maximum amount of slots at once. In general, using small sub-jobs leads to minor performance decrease but helps embedding the automatically generated screening sub-jobs into existing cluster environments.

6.2.2 Cloud Computing Cost Analysis

Then main point of interest for stakeholders deciding whether to invest in physical infrastructure or rely on cloud services is cost. We have seen that different types of virtual machine instances offered by AWS vary in their applicability for virtual screening, making it necessary to carefully choose in order to get the best virtual screening performance per dollar. The US dollar (USD) is used as base currency for all further analyses, because AWS settles all bills in USD, the same holds true also for other large cloud providers.

Table 6.3: AWS instance types used for benchmarking.

Instance Type	**Instance Cost/h ($)**	**Maximum Throughput (compounds/s)**	**#Compounds/$**
m4.large	0.12	66	1980000
m4.xlarge	0.24	129	1935000
m4.2xlarge	0.48	234	1755000
m4.4xlarge	0.96	448	1680000
c4.large	0.114	79	**2494736**
c4.xlarge	0.227	139	2204405
c4.2xlarge	0.454	279	2212334
c4.4xlarge	0.909	**492**	1948514

For further evaluation, we introduce the notion of *compounds screened per dollar*. The metric can easily be computed for a cluster by using the formula

$$\frac{throughput * 3600}{cost/h} \,. \tag{6.5}$$

As has already been mentioned, throughput is defined as screened compounds per second. Table 6.3 lists the associated values for all individual instances that have been used

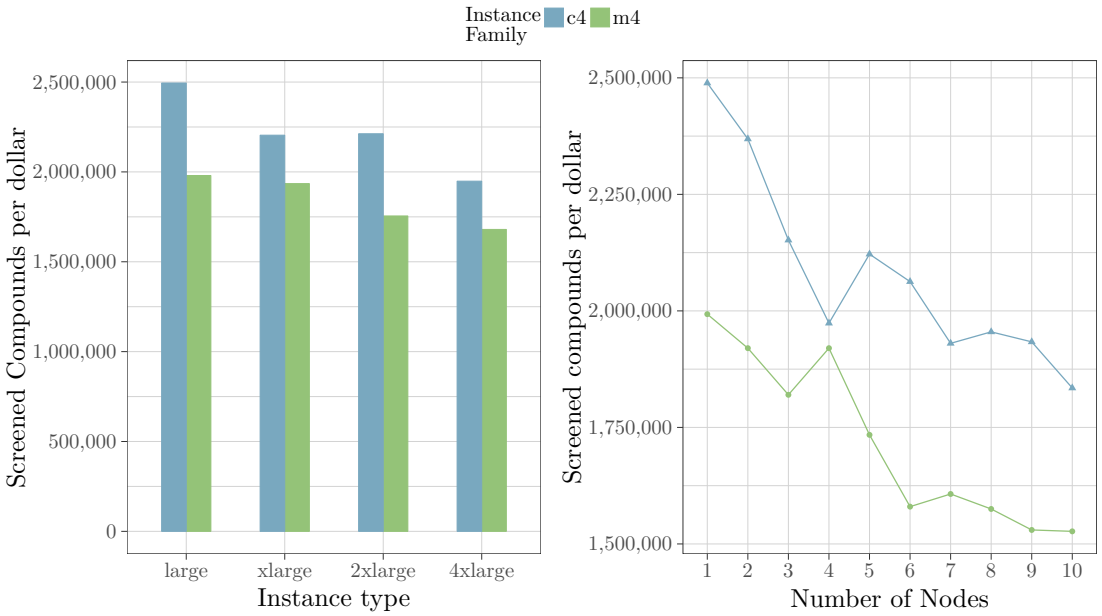

Figure 6.14: Compounds screened per dollar for different cloud instances and cluster sizes.

in Section 6.1.3. The values represent the maximum throughput that was measured as average in any 1-minute interval during one of the above experiments. As already explained, the raw values have little meaning but are primarily used for comparison. The maximum throughput value should reflect the performance that can be achieved under full load. It is better suited for comparison than the average, because it is not affected by the fluctuating overhead at the beginning of a job. This overhead depends on parameters of the job-splitting algorithm and has stronger implications on larger instances, which would hinder comparison between different instance types. Figure 6.14 provides a visual representation of these values. The illustration also shows how the cost efficiency decreases when nodes are added to a cloud cluster. This is, because costs increase linearly, while throughput rises slightly slower. The chart shows a decrease in cost efficiency by 25 percent when comparing a single-node to a cluster of 10 instances. However, as has been shown in the previous section, for larger sub-job sizes this discrepancy is less significant.

While the cost of cloud products can easily be calculated, determining the expenditures involved in running local physical clusters is complex. This, on its own, is a significant practical advantage of HPC in the cloud. The most significant part of on-site HPC costs is the upfront purchase of physical hardware. Other, less visible cost factors are electricity consumption, maintenance cost, and space occupation. The latter can be a major factor in urban areas with high property and rental prices. All three factors can greatly vary between regions and even organizations. Another major issue is the occupancy rate of physical hardware. It is clear, that on-demand cloud clusters have an advantage in this

regard. However, the magnitude of this benefit can not be reasonably assumed for an arbitrary organization. Most real world estimates of average data center server utilization range from 5% to 20% [33, 34]. For HPC clusters, the numbers are likely higher, but will greatly differ between organizations. We feel that there can be no sensible calculation of compounds screened per dollar for on-site clusters as the choice of parameter values can easily manipulate the outcome towards both directions. Nevertheless, literature provides various models for comparing on-site cluster costs to HPC in the cloud [48, 96, 46]. The overall sentiment is that, in principle, building on-site HPC clusters is still cheaper, but lacks cloud computing benefits such as scalability, elasticity, and ease-of-use. To conclude this price comparison, we suggest a hybrid approach to cluster computing. This implies having a relatively small on-site cluster in order to achieve high average utilization. On top of that, on-demand cloud clusters can be used for exceptional computation demands. The developed integration of HPC resources into the LigandScout GUI is easily able to switch between remote clusters and can therefore fully exploit such an architecture.

6.2.3 Usability

Another goal of this thesis is to enable scientists without command-line and scripting expertise to fully exploit the superior computing capabilities of HPC clusters. This is done by transparently integrating access to remote HPC clusters into the LigandScout GUI. Evaluating the usability of said integration comes down to identifying limitations to this transparency, because assessing workflows that are already inherent to LigandScout is not part of this work.

In order to use the remote screening functionality, users of the LigandScout GUI need to do three minor additional steps, compared to the previously existing local screening workflow:

1. The most important one is the configuration of connection settings. The respective dialog is shown in Figure 5.8. For obvious reasons, this requirement cannot be mitigated.
2. After the network settings are specified correctly, the connection to the remote server has to be established by a single button click.
3. After the two above steps, starting the remote screening is almost identical to doing a local screening run. The only difference is an additional dialog that allows to set various parameters such as the job's name. All fields can be seen in Figure 5.9. As the default values for these settings are dynamically retrieved from the connected server, this step can even be skipped.

The remote screening functionality also provides various additional features, which are either not possible or not relevant for local screening. However, as it has been outlined, the basic use case of starting a virtual screening experiment differs only in three easy steps.

CHAPTER 7

Conclusion and Future Work

HPC is essential for many scientific applications. The advent of cloud computing has further increased the importance of HPC by eliminating the need for investing in an on-site compute cluster. Nevertheless, experiments that are applicable for the execution on HPC hardware are commonly done on local machines, because many scientists lack the necessary command-line expertise and are subsequently dependent on GUI applications. In this thesis, we have outlined previous approaches for solving this problem, which were mainly focused on providing web-based front-ends for distributed resource management systems (DRMSs).

We have also developed our own solution for overcoming HPC usability barriers. The result is, on the one hand, a server application that can be deployed on HPC clusters and exposes RESTful web-services for submitting computational jobs as well as for monitoring and managing these jobs. The software is capable of communicating with the Sun Grid Engine and its successors as underlying DRMSs, it keeps track of all submitted jobs and their data, such as input and output files, and it can easily be deployed on any cluster node by the use of embedded default configuration settings. On the other hand, access to HPC resources has been concretely integrated into the LigandScout molecular design software through transparent invocations of the web-service resource endpoints exposed by the described server application.

The aforementioned integration enables the execution of virtual screening experiments on remote HPC clusters without having to forgo the usability benefits of a locally installed desktop GUI application. The handling of input and output data is done transparently to the user. Furthermore, progress updates are displayed in real-time, the same way as if the job were performed on the local machine. Additionally, computational jobs can be remotely executed in the background, allowing for a continued use of the then unblocked user interface. All previously submitted jobs can be monitored and managed directly within LigandScout by the use of simple interactions such as cancelling or deleting a job. The underlying implementation takes care of transmitting a request to the remote server,

which handles any necessary communication with the DRMS and deletes or moves no longer needed data. A key feature is the possibility to load past screening experiments directly into the LigandScout GUI for examination of the results or for further modeling work on the contained data. This circumvents the need for any manual handling of output files that usually comes with using remote HPC clusters. In order to act jointly with existing cluster resource management and job scheduling policies, the server application offers various configuration settings to enable customization of the job submission process. In particular, a versatile approach for splitting virtual screening jobs into several smaller sub-jobs has been designed and implemented. This allows for utilizing all nodes of a distributed-memory cluster, even if the executed algorithm is not MPI-enabled. Starting at least a subset of the resulting sub-jobs immediately leads to quick first results that can be delivered to the client, while the remaining jobs wait until additional cluster resources are free.

The performance evaluation has shown that the introduced overhead due to network communication and the developed job-splitting approach is in the range of few seconds and therefore negligible compared to the computational work for large screening experiments. Benchmark tests have further shown that an exemplary screening experiment using a relatively small database, which took about 20 minutes on a powerful desktop computer and more than 50 minutes on a recent notebook, can be finished in 45 seconds using a HPC cluster of 15 nodes. This magnitude of performance gain, without having to manually handle the job submission process, opens up new possibilities for molecular modeling work within the LigandScout GUI. In particular, it becomes feasible to run virtual screening experiments with various different configuration settings and to use large compound databases. Another use case is activity profiling, which aims to assess the biological activity of molecules towards a potentially large set of target binding sites. As a consequence, the demand for computational power is greater than for regular virtual screening experiments which use only one query model.

Apart from the transparent GUI integration and the automated splitting of large computational jobs, supporting the deployment of HPC clusters in the cloud has been a third focus of this thesis. Using the CfnCluster toolkit, a cluster of arbitrary size can be deployed in the AWS cloud using only a single command. The server application is then automatically installed and started, resulting in a ready-to-use HPC environment. Several implementation choices have been made in order to facilitate this process as well as working with the resulting cloud clusters. Specifically, the use of embedded web servers and relational database management systems along with a ready-to-use default configuration enables an automated deployment process. The data transfer, which is potentially expensive in the context of cloud computing, is kept to a minimum by circumventing the need to upload input compound libraries. Moreover, the job-splitting algorithm is well aligned with the main benefits of cloud computing. As has been successfully evaluated, the approach scales well both horizontally and vertically. Since screening experiments are potentially split into a large amount of sub-jobs that can not all be started immediately depending on the available resources, dynamically added cluster

nodes can be exploited by already running experiments. Through this support for cloud deployments, the developed software can also be used by organizations without access to a physical cluster.

The broad scope of this thesis ranging from HPC to the specific application of virtual screening and the extension of an existing GUI application opens up various possibilities for future work. One way to extend the capabilities of the server application would be to add support for other DRMSs apart from SGE, for example, the commonly used SLURM Workload Manager. Furthermore, it is planned to add other types of computational jobs. In particular for LigandScout, conformer generation is an obvious future addition. The architecture of the developed software has been carefully designed to be extensible. Another interesting proposition is to integrate access to the server's web service API into additional GUI applications. Transparent remote execution on HPC resources could make mobile devices, such as tablets, a viable alternative for scientific work. In order to further improve working with dynamic cloud environments, features to control scaling or even to create and delete on-demand clusters could be built directly into the GUI of LigandScout or other applications. The direction of this future work will be driven by reactions to this thesis and feedback from users of the developed software.

APPENDIX A

Software Versions

Table A.1: Version numbers of the used software tools.

Name	Version	Comment
Java (JDK)	8u151	Primary programming language
Spring Boot	1.5.9	Server-side application framework
Spring	4.3	The used Spring Boot framework is based on this version of Spring.
Hibernate	5.0.12	Object-relational mapping (ORM) framework
Spring Data JPA	1.11.1	Enhanced Spring support for the Java Persistence API
Jersey	2.26	Library used client-side for accessing RESTFul web services.
JSch	0.1.54	Java implementation of SSH2 used for creating SSH tunnels.
Son of Grid Engine	8.1.9	SGE version installed on `hydra` and all created AWS cloud clusters.
H2 Database Engine	1.4.196	Used as embedded database engine used for automated deployments.
MySQL	6.0.6	Database engine used as alternative to H2. Mainly for testing portability.
DRMAA	2	Implementation of the specification is provided by the Son of Grid Engine 8.1.9
Maven	3.5.2	Software project management and comprehension tool.

APPENDIX B

Class Diagram of Mapped Entities

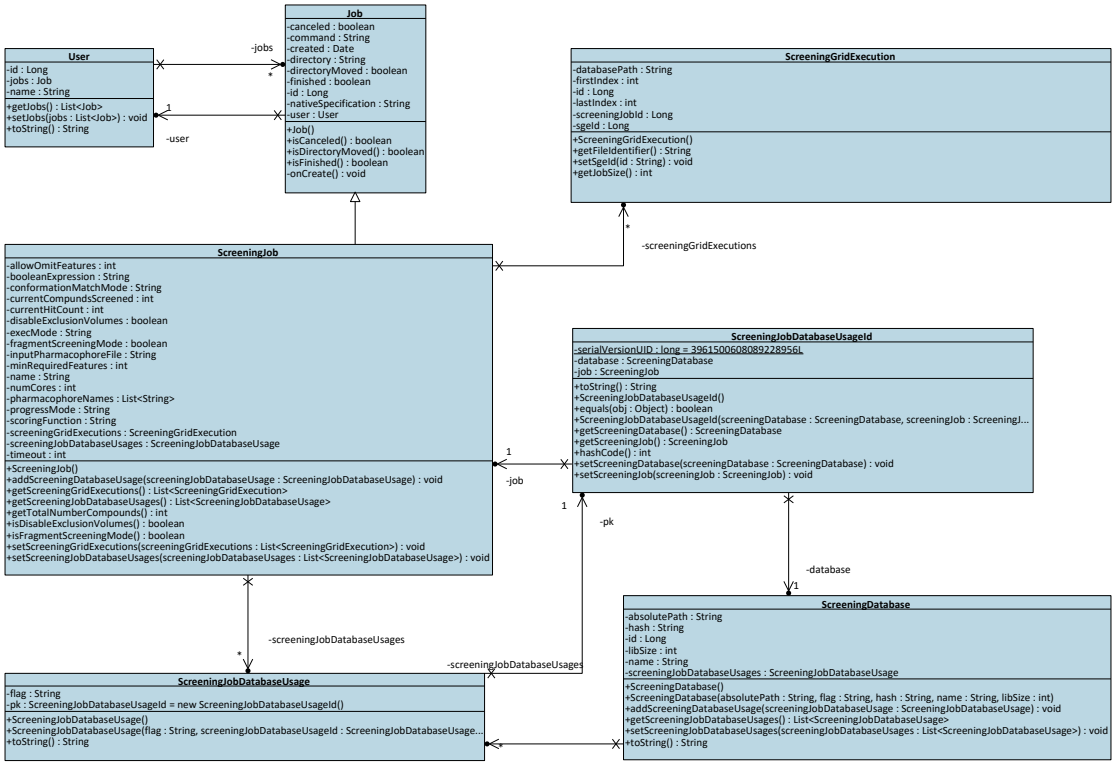

Figure B.1: Class diagram showing all entities that are mapped to database tables.

APPENDIX C

Default Server Configuration

Listing C.1 presents the default configuration that is currently shipped with the server application. The server will automatically use these for any configuration parameter that is commented out or removed completely in the user configuration file. The only parameter that has to be set in any case is `ldb.directories`, which specifies where the compound databases are located. Nevertheless, users are advised to adjust the configuration to fit their HPC environment. Most importantly, this concerns the SGE and job-splitting properties.

Listing C.1: Default configuration of the developed server application.

```
1  ######################################################################
2  # Custom properties
3  ######################################################################
4  job.directory= ./jobs
5
6  iscreen.path.executable = ./ligandscout4/iscreen
7  libsize.path.executable = ./ligandscout4/libsize
8
9  #directories considered by the monitoring process, which looks for ↩
       newly added compound databases (.ldb format required)
10 ldb.directories = <path to folder containing .ldb compound ↩
       databases>
11 ldb.directories.monitoring.recursive = true
12
13
14 ######################################################################
15 # SGE and job-splitting properties
16 ######################################################################
17 sge.parallel.environment = default
18 sge.amount.slots = 2
19 #Priority is an integer in the range -1023 to 1024.
```

```
20 #Setting a value above 0 will only be possible if the server user ↩
       is a SGE administrator or operator
21 sge.priority = 0
22
23 #Any other options that should be passed for every SGE job
24 sge.option.string=
25
26 # iScreen
27 # Amount of threads that the icreen tool should start
28 iscreen.amount.cores = 2
29 # memory value of 0 or below will lead to default iscreen setting ↩
       being used
30 iscreen.memory = 0
31
32 # Maximum mount of compounds to be screened by each sub-job. ↩
       Smaller number yields more sub-jobs.
33 job.splitting.max.chunk.size = 10000
34
35 #Data Management
36 move_finished_jobs=false
37 # Directory to which finished and cancelled jobs will be moved if ↩
       move_finished_jobs is set to true.
38 # The <user> placeholder will be replaced with the name of the user↩
        who started the job.
39 # Note that the server application has to be able to write to the ↩
       respective directorie(s).
40 finished_jobs_directory=/home/<user>/jobs
41
42
43 ########################################################################
44 #Database properties
45 ########################################################################
46 spring.datasource.url=jdbc:h2:./database/ilib-server;AUTO_SERVER=↩
       TRUE;MVCC=true
47 spring.datasource.driverClassName=org.h2.Driver
48 spring.datasource.username=<db-user>
49 spring.datasource.password=<db-password>
50 spring.jpa.generate-ddl=true
51 spring.jpa.show-sql=false
52 spring.jpa.hibernate.ddl-auto=update
53 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
54 spring.jpa.hibernate.use-new-id-generator-mappings=true
55
56
57 ########################################################################
58 # Tomcat properties
59 ########################################################################
60 server.tomcat.basedir=./tomcat-logs
61 server.tomcat.accesslog.enabled=true
```

```
62 server.tomcat.accesslog.pattern=%t %a "%r" %s (%D ms)
63
64 server.port = 8080
```

APPENDIX D

Exemplary JSON Messages

This section presents exemplary JSON messages for some of the server application's most important resource endpoints. Listing 5.1 already showed the JSON representation of a job status object. The resource endpoint `/jobs/screening/<userName>/statuses` essentially returns a list of such objects. Listing D.1 illustrates the message that is requested from the server in order to get the default parameter values for the job-splitting algorithm. These values can then be overridden on a per-request basis. An exemplary request body for starting a new screening job, which does override some of the job-splitting parameters, is given in Listing D.2. The results of an exemplary call to the `/databases` endpoint are given in Listing D.3. The response from the `/jobs/screening/<jobId>/session` endpoint, illustrated in Listing D.4, combines a job status object with information about the databases used in the requested job.

Listing D.1: Exemplary response of a call to the web service resource endpoint located at URI `jobs/screening/submissionDetails`

```
{
    "iscreenCores": 4,
    "iscreenMemory": 6,
    "maxChunkSize": 10000,
    "negativePriority": 0,
    "sgeOptionString": "",
    "sgeSlotsPerJob": 2
}
```

Listing D.2: Request body of a call to `/jobs/screening` using the HTTP verb `POST`.

```
{
   "jobName":"screening-job",
   "username":"tkainrad",
   "booleanExpression":null,
   "pharmacophoreNames":[
      "(1KE7) [A] LS3299"
```

```
   ],
   "screeningDatabasePropertiesDTOs":[
      {
         "flag":"active",
         "hash":"C1A9FE37B8F5975CB6D36CE093321BCE",
         "name":"1ke7_actives.ldb",
         "size":0
      },
      {
         "flag":"active",
         "hash":"ACFE808B7325FE633935CC6841DDF0C4",
         "name":"1ke7_decoys.ldb",
         "size":0
      }
   ],
   "screeningSettingsDTO":{
      "allowOmitFeatures":0,
      "conformationMatchMode":"FIRST",
      "disableExclusionVolumes":false,
      "enableFragmentScreeningMode":false,
      "minRequiredFeatures":3,
      "scoringFunction":"absolute",
      "screeningDatabasePropertiesDTOs":null,
      "timeOut":-1
   },
   "screeningSubmissionDetailsDTO":{
      "maxChunkSize":10000,
      "negativePriority":0,
      "sgeOptionString":null,
      "sgeSlotsPerJob":1,
      "iscreenCores":1,
      "iscreenMemory":0
   }
}
```

Listing D.3: Exemplary response of a call to the `/databases` web service resource endpoint.

```
{
    "screeningDatabases": [
        {
            "flag": "active",
            "hash": "ACFE808B7325FE633935CC6841DDF0C4",
            "name": "ZMD_13062016_filtered_chunk1.ldb",
        },
        {
            "flag": "active",
            "hash": "1EB3AD7DA286D6732410D0C2E10DAF44",
            "name": "ZMD_13062016_filtered_chunk2.ldb",
        },
        {
            "flag": "active",
            "hash": "0B29B8A86BE4062336BED2EDB291BB84",
```

```
            "name": "scubidoo_M.ldb",
        }
    ]
}
```

Listing D.4: Exemplary response of a call to the `jobs/screening/<jobId>/session` web service resource endpoint.

```
{
    "screeningDatabasePropertiesDTOs": [
        {
            "flag": "active",
            "hash": "C1A9FE37B8F5975CB6D36CE093321BCE",
            "name": "1ke7_actives.ldb",
            "size": 7
        },
        {
            "flag": "active",
            "hash": "ACFE808B7325FE633935CC6841DDF0C4",
            "name": "1ke7_decoys.ldb",
            "size": 49
        }
    ],
    "booleanExpression": "(1 and 2)",
    "jobStatus": {
        "created": 1524044525000,
        "finished": true,
        "finishedTimestamp": 1524044548000,
        "internadId": 2,
        "name": "job_2",
        "compoundsScreened": 56,
        "currentHitCount": 0,
        "errorMessage": "",
        "eta": 0,
        "pharmacophoreNames": null,
        "screeningDatabases": null,
        "screeningSettings": null,
        "sgeExecutions": [
            {
                "progress": 100,
                "sgeId": 22838,
                "sgeStatus": "Done"
            },
            {
                "progress": 100,
                "sgeId": 22839,
                "sgeStatus": "Done"
            }
        ],
        "totalAmountCompounds": 56
    }
}
```

APPENDIX E

CfnCluster Configuration

Listing E.1 gives an exemplary CfnCluster configuration file that facilitates the creation of a basic cloud cluster with two `c4.large` nodes. Naturally, placeholders are given instead of AWS credentials information.

Listing E.1: Exemplary CfnCluster configuration file.

```
1  [aws]
2  aws_region_name = eu-central-1
3  aws_access_key_id = <aws key id>
4  aws_secret_access_key = <aws secret key>
5
6  [cluster default]
7  vpc_settings = public
8  key_name = cluster-01
9  ebs_settings = custom
10 post_install = <hyperlink to web-hosted post install script>
11 base_os = ubuntu1604
12 master_instance_type = t2.micro
13 compute_instance_type = c4.large
14 scheduler = sge
15 initial_queue_size = 2
16 maintain_initial_size = true
17
18 [vpc public]
19 master_subnet_id = <vpc subnet id>
20 vpc_id = <vpc id>
21
22 [global]
23 update_check = true
24 sanity_check = true
25 cluster_template = default
26
```

```
27 [ebs custom]
28 ebs_snapshot_id = <EBS snapshot id>
29 volume_type = gp2
30 volume_size = 20
```

Listing E.2 contains the *post install script* that is executed on all cluster nodes. It installs Java and prepares the default SGE installation to be used with DRMAA. Additionally, it already activates the LigandScout installation that is present on the shared volume storage and starts the server application.

Listing E.2: CfnCluster post install script

```
1  sudo apt-get install openjdk-8-jdk openjdk-8-demo openjdk-8-doc openjdk-8-↩
       jre-headless libjemalloc-dev -y
2  sudo chown ubuntu:ubuntu -R /shared/
3
4
5  if [ "\$cfn_node_type" == "MasterServer" ]; then
6    sudo mv /opt/sge/lib/ /opt/sge/libbckup
7    sudo cp -r /shared/lib/ /opt/sge/
8    sudo chmod 755 /opt/sge/lib -R
9    su - ubuntu -c '/shared/ligandscout4/ligandscout_activation -s <↩
         LigandScout license key>'
10   su - ubuntu -c /shared/ilib-server/start-server.sh
11 fi
```

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] V. Mauch, M. Kunze, and M. Hillenbrand, "High performance cloud computing," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1408–1416, 2013. [Online]. Available: https://doi.org/10.1016/j.future.2012.03.011

[2] D. K. Brown, D. L. Penkler, T. M. Musyoka, and Ö. T. Bishop, "JMS: An open source workflow management system and web-based cluster front-end for high performance computing," *PLOS ONE*, vol. 10, no. 8, pp. 1–25, 08 2015. [Online]. Available: https://doi.org/10.1371/journal.pone.0134273

[3] G. Misra, S. Agrawal, N. Kurkure, S. Pawar, and K. Mathur, "CHReME: A web based application execution tool for using HPC resources," *International Conference on High Performance Computing (HPC-UA 2011)*, vol. 2011, pp. 12–14, 2011.

[4] A. Hunter, A. B. Macgregor, T. O. Szabó, C. A. Wellington, and M. I. Bellgard, "Yabi: An online research environment for grid, high performance and cloud computing," *Source Code for Biology and Medicine*, vol. 7, p. 1, 2012. [Online]. Available: https://doi.org/10.1186/1751-0473-7-1

[5] G. Wolber and T. Langer, "Ligandscout: 3-D pharmacophores derived from protein-bound ligands and their use as virtual screening filters," *Journal of Chemical Information and Modeling*, vol. 45, no. 1, pp. 160–169, 2005. [Online]. Available: https://doi.org/10.1021/ci049885e

[6] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[7] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," 1994. [Online]. Available: https://dl.acm.org/citation.cfm?id=898758

[8] A. Middleton and A. Chala, "HPCC systems: Introduction to HPCC (high-performance computing cluster)," *White paper, LexisNexis Risk Solutions*, 2011.

[9] N. Sadashiv and S. M. Kumar, "Cluster, grid and cloud computing: A detailed comparison," *Proceedings of the 6th International Conference on Computer Science and Education (ICCSE 2011)*, pp. 477–482, 2011. [Online]. Available: https://doi.org/10.1109/ICCSE.2011.6028683

[10] K. Krauter, R. Buyya, and M. Maheswaran, "A taxonomy and survey of grid resource management systems for distributed computing," *Software-Practice & Experience*, vol. 32, pp. 135–164, 2002. [Online]. Available: https://doi.org/10.1002/spe.432

[11] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[12] Y. Etsion and D. Tsafrir, "A short survey of commercial cluster batch schedulers," *The Hebrew University of Jerusalem, Technical Report*, vol. 13, 2005. [Online]. Available: http://leibniz.cs.huji.ac.il/tr/742.pdf

[13] Adaptive Computing, Inc., "Torque Resource Manager," Accessed on 2017-05-10. [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque/

[14] G. Staples, "TORQUE - TORQUE resource manager," in *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC 2006)*, 2006, p. 8. [Online]. Available: https://doi.org/10.1145/1188455.1188464

[15] P. Joshi and M. R. Babu, "OpenLava: An open source scheduler for high performance computing," in *Proceedings of the International Conference on Research Advances in Integrated Navigation Systems (RAINS)*. IEEE, 2016, pp. 1–3. [Online]. Available: https://doi.org/10.1109/RAINS.2016.7764375

[16] International Business Machines Corporation (IBM), "IBM Platform LSF," Accessed on 2017-10-03. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSETD4/product_welcome_platform_lsf.html

[17] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience," *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005. [Online]. Available: https://doi.org/10.1002/cpe.938

[18] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: simple linux utility for resource management," in *Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2003, pp. 44–60. [Online]. Available: https://doi.org/10.1007/10968987_3

[19] Univa Corporation, "Univa Grid Engine," Accessed on 2017-04-10. [Online]. Available: http://www.univa.com/products/

[20] W. Gentzsch, "Sun grid engine: Towards creating a compute power grid," in *Proceedings of the First IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, 2001, pp. 35–39. [Online]. Available: https://doi.org/10.1109/CCGRID.2001.923173

[21] S. Iqbal, R. Gupta, and Y.-C. Fang, "Planning considerations for job scheduling in HPC clusters," *Dell Power Solutions Magazine*, pp. 133–136, 2005.

[22] D. G. Feitelson and A. M. Weil, "Utilization and predictability in scheduling the IBM SP2 with backfilling," in *IPPS/SPDP*, 1998, pp. 542–546. [Online]. Available: https://doi.org/10.1109/IPPS.1998.669970

[23] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "Evaluation of job-scheduling strategies for grid computing," in *Proceedings of the First IEEE/ACM International Workshop on Grid Computing (GRID)*, ser. Lecture Notes in Computer Science, vol. 1971. Springer, 2000, pp. 191–202. [Online]. Available: https://doi.org/10.1007/3-540-44444-0_18

[24] A. M. Weil and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001. [Online]. Available: https://doi.org/10.1109/71.932708

[25] M. Drozdowski, *Scheduling for Parallel Processing*. Springer Publishing Company, Incorporated, 2009.

[26] D. Love, "Son of Grid Engine," Accessed on 2017-04-10. [Online]. Available: https://arc.liv.ac.uk/trac/SGE

[27] Scalable Logic, "Open Grid Scheduler," Accessed on 2017-04-10. [Online]. Available: http://gridscheduler.sourceforge.net/

[28] J.-P. Kauppi, J. Pajula, and J. Tohka, "A versatile software package for inter-subject correlation based analyses of fMRI," *Frontiers in Neuroinformatics*, vol. 8, p. 2, 2014. [Online]. Available: https://doi.org/10.3389/fninf.2014.00002

[29] Amazon.com, Inc., "CfnCluster GitHub project," Accessed on 2017-10-03. [Online]. Available: https://github.com/awslabs/cfncluster

[30] W. Gentzsch, "Sun Grid Engine: Towards creating a compute power grid," in *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2001, pp. 35–36. [Online]. Available: https://doi.org/10.1109/CCGRID.2001.923173

[31] H. Rajic, R. Brobst, W. Chan, F. Ferstl, J. Gardiner, A. Haas, B. Nitzberg, and J. Tollefsrud, "Distributed Resource Management Application API Specification 1.0," in *The Global Grid Forum*, 2004. [Online]. Available: https://www.ogf.org/documents/GFD.22.pdf

[32] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010. [Online]. Available: https://doi.org/10.1145/1721654.1721672

[33] L. Siegele, "Let it rise: A special report on corporate it," *Economist Newspaper*, 2008. [Online]. Available: http://www.economist.com/node/12411882

[34] K. Rangan, A. Cooke, J. Post, and N. Schindler, "The cloud wars: $100+ billion at stake," *Technical Report, Merrill Lynch*, 2008.

[35] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, vol. 51, no. 1, pp. 137–149, 2004. [Online]. Available: https://doi.org/10.1145/1327452.1327492

[36] T. White, *Hadoop: The definitive guide*. O'Reilly Media, 2012.

[37] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10. [Online]. Available: https://doi.org/10.1109/MSST.2010.5496972

[38] Amazon.com, Inc., "Amazon Elastic MapReduce," Accessed on 2017-10-03. [Online]. Available: https://aws.amazon.com/en/emr/

[39] Microsoft Corporation, "Microsoft Azure HDInsight," Accessed on 2017-10-03. [Online]. Available: https://azure.microsoft.com/us-en/services/hdinsight/

[40] Google Inc., "Google Cloud Platfrom Cloud Dataproc," Accessed on 2017-10-03. [Online]. Available: https://cloud.google.com/dataproc/

[41] Amazon.com, Inc., "AWS Batch," Accessed on 2017-10-03. [Online]. Available: https://aws.amazon.com/en/batch/

[42] M. de Bayser and R. F. G. Cerqueira, "Integrating MPI with Docker for HPC," in *Proceedings of the International Conference on Cloud Engineering (IC2E)*. IEEE, 2017, pp. 259–265. [Online]. Available: https://doi.org/10.1109/IC2E.2017.40

[43] Massachusetts Institute of Technology (MIT), "StarCluster," Accessed on 2017-10-03. [Online]. Available: http://star.mit.edu/cluster/

[44] University of Zurich, "ElastiCluster GitHub project," Accessed on 2017-10-03. [Online]. Available: https://github.com/gc3-uzh-ch/elasticluster

[45] S. Haug, F. Sciacca, and ATLAS Collaboration, "Atlas computing on Swiss cloud SWITCHengines," in *Journal of Physics: Conference Series*, vol. 898, no. 5. IOP Publishing, 2017, p. 052017. [Online]. Available: https://doi.org/10.1088/1742-6596/898/5/052017

[46] A. Marathe, R. Harris, D. K. Lowenthal, B. R. de Supinski, B. Rountree, M. Schulz, and X. Yuan, "A comparative study of high-performance computing on the cloud," in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2013, pp. 239–250. [Online]. Available: https://doi.acm.org/10.1145/2462902.2462919

[47] M. Kang, D. Kang, J. P. Walters, and S. P. Crago, "A comparison of system performance on a private openstack cloud and Amazon EC2," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, June 25-30, 2017*. IEEE Computer Society, 2017, pp. 310–317. [Online]. Available: https://doi.org/10.1109/CLOUD.2017.47

[48] J. Napper and P. Bientinesi, "Can cloud computing reach the TOP500?" in *Proceedings of the Combined Workshops on Unconventional High Performance Computing Workshop Plus Memory Access Workshop*, ser. UCHPC-MAW '09. ACM, 2009, pp. 17–20. [Online]. Available: https://doi.org/10.1145/1531666.1531671

[49] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003. [Online]. Available: https://doi.org/10.1002/cpe.728

[50] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the Amazon web services cloud," in *Proceedings of the Second International Conference on Cloud Computing (CloudCom)*. IEEE Computer Society, 2010, pp. 159–168. [Online]. Available: https://doi.org/10.1109/CloudCom.2010.69

[51] Amazon.com, Inc., "AWS Novartis Case Study," Accessed on 2017-10-03. [Online]. Available: https://aws.amazon.com/solutions/case-studies/novartis/

[52] Cycle Computing, "Hgst buys 70,000-core cloud hpc cluster, breaks record, returns it 8 hours later," Accessed on 2017-10-03. [Online]. Available: https://cyclecomputing.com/hgst-buys-70000-core-cloud-hpc-cluster-breaks-record-8-hours/

[53] G. Wolber and T. Langer, "Ligandscout: 3-d pharmacophores derived from protein-bound ligands and their use as virtual screening filters," *Journal of Chemical Information and Modeling*, vol. 45, no. 1, pp. 160–169, 2005. [Online]. Available: https://doi.org/10.1021/ci049885e

[54] C. G. Wermuth, C. R. Ganellin, P. Lindberg, and L. A. Mitscher, "Glossary of terms used in medicinal chemistry (IUPAC Recommendations 1998)," *Pure and Applied Chemistry*, vol. 70, no. 5, pp. 1129–1143, Jan. 1998. [Online]. Available: https://doi.org/10.1351/pac199870051129

[55] B. K. Shoichet, "Virtual screening of chemical libraries." *Nature*, vol. 432, no. 7019, pp. 862–5, dec 2004. [Online]. Available: https://doi.org/10.1038/nature03197

[56] D. B. Kitchen, H. Decornez, J. R. Furr, and J. Bajorath, "Docking and scoring in virtual screening for drug discovery: methods and applications," *Nature Reviews Drug Discovery*, vol. 3, no. 11, pp. 935–949, nov 2004. [Online]. Available: https://doi.org/10.1038/nrd1549

[57] D. Schuster, "3D pharmacophores as tools for activity profiling," *Drug Discovery Today: Technologies*, vol. 7, no. 4, pp. e205–e211, dec 2010. [Online]. Available: https://doi.org/10.1016/j.ddtec.2010.11.006

[58] T. Seidel, G. Ibis, F. Bendix, and G. Wolber, "Strategies for 3D pharmacophore-based virtual screening," *Drug Discovery Today: Technologies*, vol. 7, no. 4, pp. e221–e228, 2010, 3D Pharmacophore Elucidation and Virtual Screening. [Online]. Available: https://doi.org/10.1016/j.ddtec.2010.11.004

[59] N. Triballeau, F. Acher, I. Brabet, J.-P. Pin, and H.-O. Bertrand, "Virtual screening workflow development guided by the 'receiver operating characteristic' curve approach. Application to high-throughput docking on metabotropic glutamate receptor subtype 4," *Journal of Medicinal Chemistry*, vol. 48, no. 7, pp. 2534–2547, 2005, pMID: 15801843. [Online]. Available: https://doi.org/10.1021/jm049092j

[60] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* Elsevier/Morgan Kaufmann, 2008.

[61] G. D. Guerrero, H. E. P. Sánchez, J. M. Cecilia, and J. M. García, "Parallelization of virtual screening in drug discovery on massively parallel architectures," in *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).* IEEE, 2012, pp. 588–595. [Online]. Available: https://doi.org/10.1109/PDP.2012.26

[62] Adaptive Computing, Inc., "Moab HPC Suite," Accessed on 2017-05-11. [Online]. Available: http://www.adaptivecomputing.com/products/hpc-products/

[63] A. Hunter, A. B. Macgregor, T. O. Szabó, C. A. Wellington, and M. I. Bellgard, "Yabi: An online research environment for grid, high performance and cloud computing," *Source Code for Biology and Medicine*, vol. 7, p. 1, 2012. [Online]. Available: https://doi.org/10.1186/1751-0473-7-1

[64] Centre for Comparative Genomics, "Yabi GitHub project," Accessed on 2017-10-03. [Online]. Available: https://github.com/muccg/yabi

[65] Brown, David K. AND Penkler, David L. AND Musyoka, Thommas M. AND Bishop, Özlem Tastan, "JMS GitHub project," Accessed on 2017-10-03. [Online]. Available: https://github.com/RUBi-ZA/JMS

[66] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences." *Genome biology*, vol. 11, no. 8, p. R86, 2010. [Online]. Available: https://doi.org/10.1186/gb-2010-11-8-r86

[67] T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004. [Online]. Available: https://doi.org/10.1093/bioinformatics/bth361

[68] K. Wolstencroft, R. Haines, D. Fellows, A. R. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. N. de la Hidalga, M. P. B. Vargas, S. Sufi, and C. A. Goble, "The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic Acids Research*, vol. 41, no. Webserver-Issue, pp. 557–561, 2013. [Online]. Available: https://doi.org/10.1093/nar/gkt328

[69] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, vol. 35, no. 4, pp. 316–319, 2017. [Online]. Available: https://doi.org/10.1038/nbt.3820

[70] Autodesk Corporation, "AutoCAD Product Page," Accessed on 2017-10-03. [Online]. Available: https://www.autodesk.com/products/autocad/overview

[71] Schrödinger, LLC, "Maestro Product Page," Accessed on 2017-10-03. [Online]. Available: https://www.schrodinger.com/maestro

[72] S. Vinoski, "Corba: Integrating diverse applications within distributed heterogeneous environments," *Comm. Mag.*, vol. 35, no. 2, pp. 46–55, Feb. 1997. [Online]. Available: https://doi.org/10.1109/35.565655

[73] Object Management Group, *The Common Object Request Broker (CORBA): Architecture and Specification.* Object Management Group, 1995.

[74] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. 'big' web services: making the right architectural decision," in *Proceedings of the 17th International Conference on World Wide Web, (WWW).* ACM, 2008, pp. 805–814. [Online]. Available: https://doi.org/10.1145/1367497.1367606

[75] Amazon.com, Inc., "Amazon Simple Storage Service Documentation," Accessed on 2017-10-04. [Online]. Available: https://aws.amazon.com/documentation/s3/

[76] M. Davies, M. Nowotka, G. Papadatos, N. Dedman, A. Gaulton, F. Atkinson, L. J. Bellis, and J. P. Overington, "ChEMBL web services: streamlining access to drug discovery data and utilities," *Nucleic Acids Research*, vol. 43, no. Webserver-Issue, pp. W612–W620, 2015. [Online]. Available: https://doi.org/10.1093/nar/gkv3522

[77] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures.* University of California, Irvine Doctoral dissertation, 2000.

[78] L. Masinter, T. Berners-Lee, and R. T. Fielding, "Uniform Resource Identifier (URI): Generic Syntax." [Online]. Available: https://tools.ietf.org/html/rfc3986

[79] L. Richardson and S. Ruby, *RESTful web services.* O'Reilly Media, Inc., 2008.

[80] International Telecomunication Union (ITU), "Information technology – open systems interconnection – basic reference model: The basic model," *ITU-T X.200*, 1994.

[81] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, vol. 35, no. 4, pp. 316–319, 2017. [Online]. Available: https://doi.org/10.1038/nbt.3820

[82] R. Rivest, "The MD5 Message-Digest Algorithm." [Online]. Available: https://tools.ietf.org/html/rfc1321

[83] V. Roussev, "Hashing and data fingerprinting in digital forensics," *IEEE Security & Privacy*, vol. 7, no. 2, pp. 49–55, 2009. [Online]. Available: https://doi.org/10.1109/MSP.2009.40

[84] Amazon.com, Inc., "AWS EC2 Instance Types," Accessed on 2017-09-12. [Online]. Available: https://aws.amazon.com/en/ec2/instance-types/

[85] R. Johnson, J. Hoeller, A. Arendsen, and R. Thomas, *Professional Java development with the Spring framework*. John Wiley & Sons, 2009.

[86] Pivotal Software, Inc., "Spring Framework GitHub project," Accessed on 2018-01-02. [Online]. Available: https://github.com/spring-projects/spring-framework

[87] ——, "Spring Boot GitHub project," Accessed on 2018-01-02. [Online]. Available: https://github.com/spring-projects/spring-boot

[88] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[89] X. Wang and H. Yu, "How to break MD5 and other hash functions," in *Proceedings of the 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. Lecture Notes in Computer Science, vol. 3494. Springer, 2005, pp. 19–35. [Online]. Available: https://doi.org/10.1007/11426639_2

[90] Apache Software Foundation, "Apache Derby Documentation," Accessed on 2018-01-21. [Online]. Available: https://db.apache.org/derby/docs/10.0/manuals/develop/develop13.html

[91] Oracle Corporation, "Jersey: RESTful Web Services in Java." Accessed on 2018-02-02. [Online]. Available: https://jersey.github.io/

[92] J. J. Irwin and B. K. Shoichet, "ZINC - A free database of commercially available compounds for virtual screening," *Journal of Chemical Information and Modeling*, vol. 45, no. 1, pp. 177–182, 2005. [Online]. Available: https://doi.org/10.1021/ci049714+

[93] S. Lu, S. Y. Tsai, and M.-J. Tsai, "Regulation of androgen-dependent prostatic cancer cell growth: androgen regulation of CDK2, CDK4, and CKI p16 genes," *Cancer research*, vol. 57, no. 20, pp. 4511–4516, 1997. [Online]. Available: http://cancerres.aacrjournals.org/content/57/20/4511.long

[94] O. Tetsu and F. McCormick, "Proliferation of cancer cells despite CDK2 inhibition," *Cancer cell*, vol. 3, no. 3, pp. 233–245, 2003. [Online]. Available: https://doi.org/10.1016/S1535-6108(03)00053-9

[95] M. M. Mysinger, M. Carchia, J. J. Irwin, and B. K. Shoichet, "Directory of useful decoys, enhanced (DUD-E): Better ligands and decoys for better benchmarking," *Journal of Medicinal Chemistry*, vol. 55, no. 14, pp. 6582–6594, 2012, pMID: 22716043. [Online]. Available: https://doi.org/10.1021/jm300687e

[96] A. G. Carlyle, S. L. Harrell, and P. M. Smith, "Cost-effective HPC: the community or the cloud?" in *Proceedings of the Second International Conference on Cloud Computing (CloudCom)*. IEEE Computer Society, 2010, pp. 169–176. [Online]. Available: https://doi.org/10.1109/CloudCom.2010.115