Data Management System for Distributed Virtual Screening

Ting Zhou and Amedeo Caflisch*

Department of Biochemistry, University of Zürich, Winterthurerstrasse 190, CH-8057 Zürich, Switzerland

Received August 22, 2008

High throughput docking (HTD) using high performance computing platforms is a multidisciplinary challenge. To handle HTD data effectively and efficiently, we have developed a distributed virtual screening data management system (DVSDMS) in which the data handling and the distribution of jobs are realized by the open-source structured query language database software MySQL. The essential concept of DVSDMS is the separation of the data management from the docking and ranking applications. DVSDMS can be used to dock millions of molecules effectively, monitor the process in real time, analyze docking results promptly, and process up to 10^8 poses by energy ranking techniques. In an HTD campaign to identify kinase inhibitors a low cost Linux PC has allowed DVSDMS to efficiently assign the workload to more than 500 computing clients. Notably, in a stress test of DVSDMS that emulated a large number of clients, about 60 molecules per second were distributed to the clients for docking, which indicates that DVSDMS can run efficiently on very large compute cluster (up to about 40000 cores).

INTRODUCTION

In silico screening of large libraries of compounds is a commonly used tool in drug discovery because it efficiently identifies candidate lead compounds.[1−7] Its efficiency is due to methodological progresses and the ever increasing performance of ordinary low-cost computers. Despite these progresses, handling large libraries of compounds is still a challenge for data management in drug design and discovery. The demand for an efficient data management increases even further if multiple computing instances access and update the data simultaneously.

Recently, several applications of large-scale virtual screening in parallel have been reported. For instance, para_glide, a recently developed extension of Glide[8,9] for parallel execution, counts the number of ligands, divides them into equal segments, and distributes them over several processors or machines. At the end of all docking calculations it provides a unified report of docking scores. Zhang and co-workers have developed the free package DOVIS which runs AutoDock[10] in parallel.[11] With DOVIS users can submit multiple jobs from a graphical user interface to both cluster and standalone computers. The authors docked about 2 million compounds on a Linux cluster with 256 CPUs and observed near-optimal performance. The essential concept of para_glide and DOVIS is the splitting of the molecular database into multiple partitions, which are then submitted to different processors individually, and the results are retrieved from all processors and combined after docking. In this way, the number of partitions and the amount of molecules are determined before splitting, the running time of each partition cannot be estimated, and the balance of each processor cannot be guaranteed either. Moustakas and Kuntz developed the MPI version of DOCK,[12] which used a master-worker scheme for parallelization.[13] To reduce bookkeeping tasks associated with manual partition of jobs, data were distributed to workers as molecules were read by the master, such that the poor load balance due to the random distribution of jobs was circumvented. Furthermore, Peters and co-workers recently optimized and validated DOCK on a massively parallel system with more than 16000 processors.[14] They pointed out that as the number of processors increased, the HTC (High Throughput Computing)[15] version of the DOCK program was more efficient than the MPI version, since library docking could be run as a collection of independent tasks while the MPI version suffered from overloading of the master. In other words, the efficiency of distribution of the master is the bottleneck of the master-worker scheme, in particular when a considerable amount of workers request jobs simultaneously. As an example, the efficiency of the MPI version of DOCK is 88% at 8192 workers but decreases to 55% at 16384 workers.[14] The overloading has been overcome by employing multilevel master-worker scheme (MLMW). However, both HTC and MLMW require additional time-consuming programming, in particular, HTC demands for the implementation of an asynchronous task dispatch subsystem, while MLMW requires the modification of the source code of the docking software.

The efficiency of data management is crucial in parallel applications. Furthermore, there is a strong demand of an efficient and easy-to-implement procedure to handle the data for a large number of computing clients. At present, most docking software reads the input and stores the output in plain files directly. Nonetheless, storing massive data in plain files is not suitable for extensive data management, since it usually requires more application programming effort to create, modify, and access data efficiently and securely. A database management system is a computer software designed to handle massive data efficiently. Providing controls of communication and synchronization, it allows multiple tasks to access and update the data in parallel with marginal

* Corresponding author phone: (+41 44) 635 55 21; fax: (+41 44) 635 68 62; e-mail: caflisch@bioc.uzh.ch.

10.1021/ci800295q CCC: $40.75
Published on Web 12/12/2008

Table 1. Structure of Table "ZINCMOL"[a]

column name	data type	explanation
zincmol.id	int(11) unsigned not null auto_increment	a unique identity for each molecule
zincmol.numatoms	tinyint(3) unsigned not null	number of atoms
zincmol.numc	tinyint(3) unsigned not null	number of carbon atoms
zincmol.numn	tinyint(3) unsigned not null	number of nitrogen atoms
zincmol.numo	tinyint(3) unsigned not null	number of oxygen atoms
zincmol.numhal	tinyint(3) unsigned not null	number of halogen atoms
zincmol.nums	tinyint(3) unsigned not null	number of sulfur atoms
zincmol.nump	tinyint(3) unsigned not null	number of phosphorus atoms
zincmol.numarombnd	tinyint(3) unsigned not null	number of aromatic bonds
zincmol.numdoubbnd	tinyint(3) unsigned not null	number of double bonds
zincmol.numtribnd	tinyint(3) unsigned not null	number of triple bonds
zincmol.numamibnd	tinyint(3) unsigned not null	number of amide bonds
zincmol.numacc	tinyint(3) unsigned not null	number of hydrogen bond acceptors
zincmol.numdon	tinyint(3) unsigned not null	number of hydrogen bond donors
zincmol.numring	tinyint(3) unsigned not null	number of rings
zincmol.totringsize	tinyint(3) unsigned not null	number of heavy atoms in rings
zincmol.longestchain	tinyint(3) unsigned not null	longest chain of atoms in the molecule
zincmol.wienerind4	float(16,14) not null	Wiener index
zincmol.numbnd	tinyint(3) unsigned not null	number of bonds
zincmol.numfrg	tinyint(3) unsigned not null	number of fragments
zincmol.numrotbnd	tinyint(3) unsigned not null	number of rotatable bonds
zincmol.mw	float(8,3) unsigned not null	molecule weight
zincmol.clogp	float(5,2) not null	CLogP
zincmol.charge	int(2) not null	formal charge
zincmol.mol2file	blob	compressed mol2 file
zincmol.tag1	char(20) default null	notes of calculation status for first target
zincmol.tag2	char(20) default null	notes of calculation status for second target
...	...	...
zincmol.tag*n*	char(20) default null	notes of calculation status for *n*th target

[a] The data types are represented in MySQL syntax.[33] The column "zincmol.id" is the primary key. The auto-incremental identifier can be used to discriminate individual protonation states and/or tautomeric forms. The 23 following columns contain the atomic and chemical properties of a molecule. The column "zincmol.mol2file" contains the compressed molecule file in mol2 format. The last columns "zincmol.tag*n*" (tag columns) record the calculation status for each protein target (e.g., multiple structures of protein or multiple proteins). Besides the primary key on column "zincmol.id", indexes are built on tag columns to speed up checking of the status by the computing clients. Other columns were not indexed because there was no query on them in the applications presented here.

additional effort in programming. It contains mature facilities to keep data integrated and consistent and provides utilities for database maintaining, such as backup, recovery, monitoring, and tuning.

In this paper, we introduce the Distributed Virtual Screening Data Management System (DVSDMS) for docking and ranking based on a master-worker scheme and the freely available database software MySQL. By separating operations related to data management from the main application and allocating the former to the database management system, DVSDMS can manage HTD data effectively and efficiently. The connection between the different programs is handled via scripts written in Python. A MySQL database, the master of the system, is used for organizing all the data. The clients (workers) do not communicate among each other but only with the database. In the application presented here, AutoDock was used as the docking engine, while DAIM,[16] Witnotp,[17] CHARMM,[18] and MOPAC[19] were employed for preparing the compound library, file type conversion, minimizing poses, and quantum mechanical calculations, respectively. Because of the flexibility of DVSDMS, other programs can be used for docking and ranking. Alternative procedures range from simple molecular mechanics energy terms and combinations thereof[20] to a quantum mechanics based ranking approach.[21] Furthermore, it is straightforward to use DVSDMS for consensus scoring.[22−24] In fact energy values calculated by different scoring functions can be stored in tables for ranking.

DVSDMS was validated in this study by docking about 1.5 million compounds into the ATP-binding site of the receptor tyrosine kinase EphB4 and ranking about 100 million poses using two Beowulf clusters (located on the same grid but with a number of switches ranging between 1 and 4). In the productive phase of docking and ranking the average load of the database management system was less than 10% and 30% with more than 500 workers requesting jobs, respectively. In a stress test, the database server built on a low cost Linux PC was able to distribute about 60 molecules per second.

DOCKING AND RANKING BY DVSDMS

The following three subsections describe briefly the overall process and programs used for docking and ranking in this application of DVSDMS. Details of the DVSDMS architecture are given in the next section.

Predocking. All structures and properties of the molecules required for docking and ranking were calculated and stored in the database. For each molecule in the ZINC library[25] (version 7) CHARMm[26] atom types were assigned with Witnotp.[17] Then DAIM[16] was applied to calculate the atomic and chemical properties of each molecule (listed in Table 1). Even though not all of these properties were used in docking and ranking, they were prepared for different kinds of filters one might want to apply before docking. Besides these properties, the mol2 file of each molecule was also

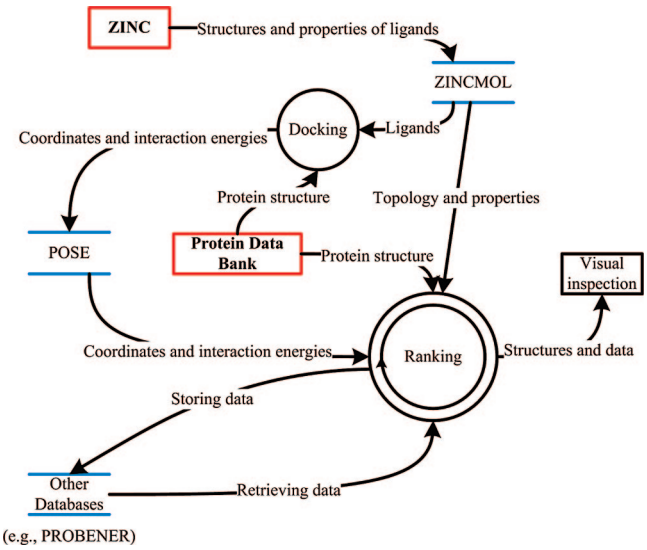

Figure 1. Schematic representation of docking and ranking processes. Red boxes indicate the public-domain databases, while blue parallels mean the tables in DVSDMS.

Table 2. Structure of Table "POSE"[a]

column name	data type	explanation
pose.id	int(11) not null auto_increment	a unique identity for each pose
pose.ele	float	electrostatic interaction
pose.elee	float	electrostatic efficiency
pose.vdw	float	vdW interaction energy
pose.vdwe	float	vdW efficiency
pose.pdbfile	blob	compressed pdb file
pose.mol_id	int(11) unsigned not null	related zincmol.id in Table "ZINCMOL"

[a] The data types are represented in MySQL syntax.[33] The Column "pose.id" is the primary key. An index is built on Column "pose.mol_id", which is a pointer for connecting the record in Table "POSE" to the one in Table "ZINCMOL". The value of Column "pose.mol_id" equals to the value of the primary key of Table "ZINCMOL".

stored in the database. A table termed "ZINCMOL" (Table 1) was designed to store these data (Figure 1).

Docking. AutoDock[10] (version 4) was applied for docking small molecules from the library into the receptor (see the Supporting Information). The poses of each molecule in the PDB format, with their interaction energies with the receptor and efficiencies (electrostatics and vdW), were stored in the table "POSE" (Table 2) of the database. During the docking, the computing clients acquired the 3D structure of the molecules directly from the database and stored poses and energies in the database after each docking process finished (Figure 1).

Ranking. Different scoring approaches can be handled efficiently by DVSDMS. In the present application to the EphB4 kinase, we used an in-house developed approach based on calculations of semiempirical quantum mechanics to efficiently rank the poses (Zhou et al., manuscript in preparation). Ranking a pose was usually faster than docking a molecule; therefore, the former needed a more efficient database I/O environment than the latter (see Results).

ARCHITECTURE OF DVSDMS

Database Structure. All the actions of the computing clients were coordinated by the database server (Figure 2). Three tables are necessary in the database for handling poses and molecules: one for the data of the molecules (Table 1), one for the poses from docking (Table 2), and one for the results of the ranking process (Table 3). Besides the properties of the molecules and mol2 files, the column "zincmol.tag" was added to Table "ZINCMOL" to record the status of docking. The format of the "zincmol.tag" column is shown in Figure 3. In the case of multiple protein targets, additional tag columns can be appended for each of the targets to record the status of docking processes. If the amount of targets is large and the appended tag columns affect the efficiency of the database, the ZINCMOL table can be vertically partitioned like the POSE table (not used in this application, see Partitioning Large Tables). The main columns, such as the properties and the coordinates of molecules, do not need to be copied for each target. If the molecule has not been handled for the specific target, the related "zincmol.tag" column is set to "null". Before the docking starts the non-drug-like molecules (according to user-defined filters) can be marked as "not passed" at "zincmol.tag" columns (see Results). The database server returns one of the molecules with a "null" tag when a docking client requests a job. The molecules which were marked as "not passed" will not be returned to computing clients. The "pose.tag" column of Table "POSE" (Table 2) was partitioned into another table ("POSETAG", Table 4) to improve the database performance (see **Partitioning Large Tables**). Column "posetag.sign" in the Table "POSETAG" is the status of ranking process of the related pose: "1", "2", "3", and "6" mean "in process", "finished normally", "finished with errors", and "unhandled", respectively. When a ranking client requests a job, the database returns a set of poses with "unhandled" signs.

Interface between Software Packages. Python scripts were developed for connecting all software packages in DVSDMS. The whole process needed the cooperation of several types of software (Table 5) developed by different groups. Therefore, some jobs such as converting file type, preparing input files, and parsing output files were needed to connect each stage of the pipeline. The communication to the database is essential in the DVSDMS. The Python package SQLAlchemy[27] is used to establish the crucial connection of python scripts (connecting the different stages of the docking pipeline) and the MySQL database (the data and result storage facility). The package Elixir[28] (object-relational mapping features) facilitates the consequent treatment of all data entries as objects, which results in clean code such that raw SQL statements are used only in performance critical parts.

Running on Standalone and Cluster Computers. DVSDMS runs as a single executable Python script on standalone or cluster computers without a queuing system. In the presence of a queuing system DVSDMS can be submitted to the queue with a runtime limit in accordance with the configuration of the queuing system. In this case, the DVSDMS client estimates the execution time of the next job before acquiring it from the database.

Monitoring Process and Identifying Errors. Users can monitor progress of the computation and trace errors by means of "sign" and "tag" columns of the corresponding tables in DVSDMS. The docking status of a molecule, machine name of the client, and the starting time of the job

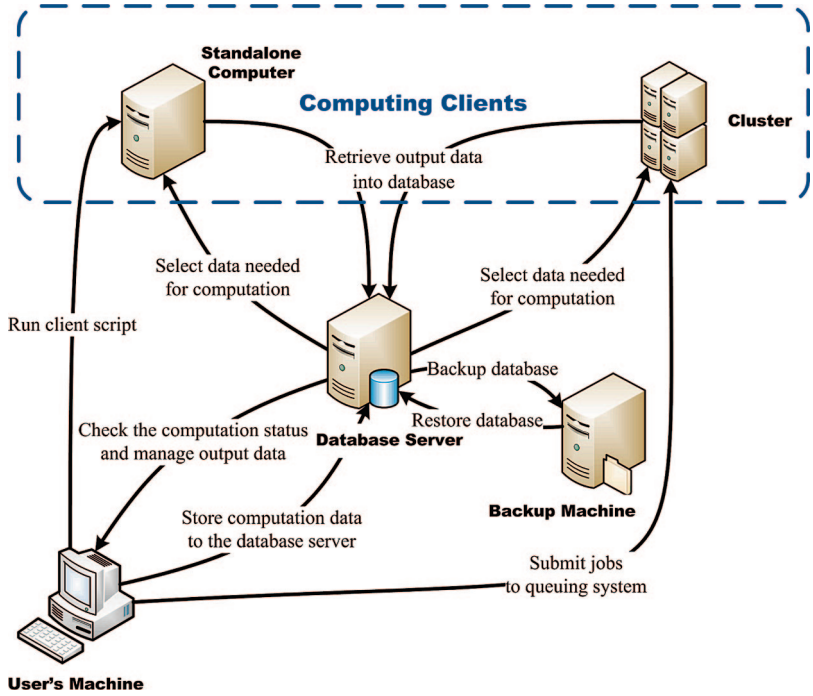

Figure 2. Hardware and data flow in DVSDMS. Note that in the application presented here the user's machine, backup machine, and database server were all on a single PC.

Table 3. Structure of Table for Ranking (PROBENER in Our Application)[a]

column name	data type	explanation
probener.id	int(11) not null auto_increment	auto increment "probener" id
probener.p*1*	float	interaction energy with the first probe
probener.p2	float	interaction energy with the second probe
...	...	...
probener.p*n*	float	interaction energy with the *n*th probe
probener.pose_id	int(11)	related pose.id in Table "POSE"

[a] The data types are represented in MySQL syntax.[33] The Column "probener.id" is the primary key. An index was built on Column "probener.pose_id", which is a pointer for connecting the record in Table "PROBENER" to the one in Table "POSE". The value of Column "probener.pose_id" equals to the value of the primary key of Table "POSE".

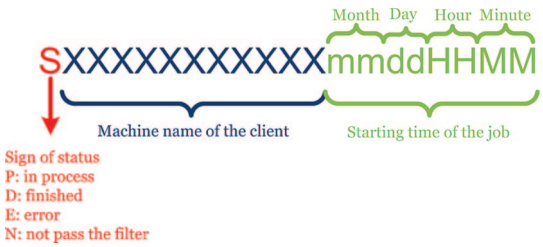

Figure 3. Format of "zincmol.tag*n*" column in Table "ZINCMOL".

can be read from Column "zincmol.tag" in Table "ZINCMOL" (Figure 3). In Table "POSETAG", Column "posetag.sign" is further separated from Column "posetag.tag" (Table 4) because in the I/O intensive ranking stage, the database only needs to scan "posetag.sign" to attain the status of the pose when requested for an "unhandled" pose.

Table 4. Structure of Table "POSETAG"[a]

column name	data type	explanation
posetag.id	int(11) not null auto_increment	auto increment posetag ID
posetag.pose_id	int(11)	related pose.id in Table POSE
posetag.sign	int(2) unsigned default '6'	sign of status
posetag.tag	char(20) default null	note for calculation status

[a] The data types are represented in MySQL syntax.[33] The Column "posetag.id" is the primary key. The Column "posetag.sign" was introduced for efficient retrieval of the ranking status. The meanings of signs are mentioned in the main text. An index was built on Column "posetag.pose_id", which is a pointer for connecting the record in Table "POSETAG" to the one in Table "POSE". The value of Column "posetag.pose_id" equals to the value of the primary key of Table "POSE".

Distributed computing systems are more prone to error than a standalone computer. It is very labor-consuming to reroll the process and locate errors when millions of compounds are handled in a high-throughput screening campaign. DVSDMS records stage information of clients in "sign" and "tag" columns and can check the status of jobs with user-defined frequency. Practically if a job does not finish during a given period of time, its status will be set to "unhandled", and its executing client will be reported. Then the job returns to the waiting list and is ready to be assigned to another free client.

PERFORMANCE TUNING

The optimization of DVSDMS focuses on the database performance tuning because, as mentioned above, the data management is separated from the main docking and ranking applications in DVSDMS. In the following, details on the process of optimization are given.

Table 5. Software and Its Function

software name	function	ref
DAIM	calculate the properties of molecules	16
AutoDockTools	convert molecule file into pdbqt type	34
AutoDock	dock small molecules into receptor	10
CHARMM	add hydrogen atoms, and minimize structure	18
Witnotp	convert molecule file types among mol2, pdb, and psf	17
MOPAC	calculate QM energies used for ranking	19
MySQL	the database management system	35
SQLAlchemy	the database toolkit for Python	27
Elixir	a declarative layer on top of SQLAlchemy	28

Using Database Index. An index was created on the column which was of high querying frequency. Each table has a primary key, a unique index to identify each row in a table, used for attaining the specific record promptly. For instance, with the primary key, any molecule record in the Table "ZINCMOL" can be retrieved out of millions of others by its ID stored in the Column "zincid.id" (Table 1) in one millisecond after the index was cached in the memory (see RESULTS section for hardware description). This is also valuable when the application needs to find out which molecule a given pose belongs to via Column "pose.mol_id" in Table "POSE" (Table 2). Similarly, but in an inverse way, the index of Column "pose.mol_id" (Table 2) is of use for fast reverse query, e.g., attaining poses related to a molecule. Only columns frequently queried are indexed. Other columns, e.g., interaction energies of poses and ranking scores, are not indexed because each additional index increases the size of the database and reduces the writing speed of tables.

Partitioning Large Tables. The performance of the database can be improved by partitioning large tables. During the database scan operation initiated by a query, only partitions containing the data are accessed, and during the maintenance, only damaged partitions instead of the entire table are repaired. Furthermore, the partitioned tables can be distributed on different physical drives, and tables can be scanned in parallel to improve both CPU and disk performance (which was not necessary for the present application). Two major forms of partitioning were applied in our database:

Horizontal Partitioning: Tables "ZINCMOL" and "POSE" were horizontally segmented into 50 partitions according to the hash function of their primary keys. Partitioning by hash is used primarily to ensure an even distribution of data among a predetermined number of partitions.[29] The value of a hash function determines the membership of a partition, e.g., the hash function returns an integer from 0 to 49 in the case with 50 partitions. The horizontal partitioning feature is supported by MySQL starting from version 5.1.

Vertical Partitioning: The most referenced columns "sign" and "tag" in the original Table "POSE" were separated into a new table "POSETAG" (Figure 4). Table "POSE" and "POSETAG" referred to each other via the column "pose.id" in "POSE" and the column "posetag.pose_id" in "POSETAG".

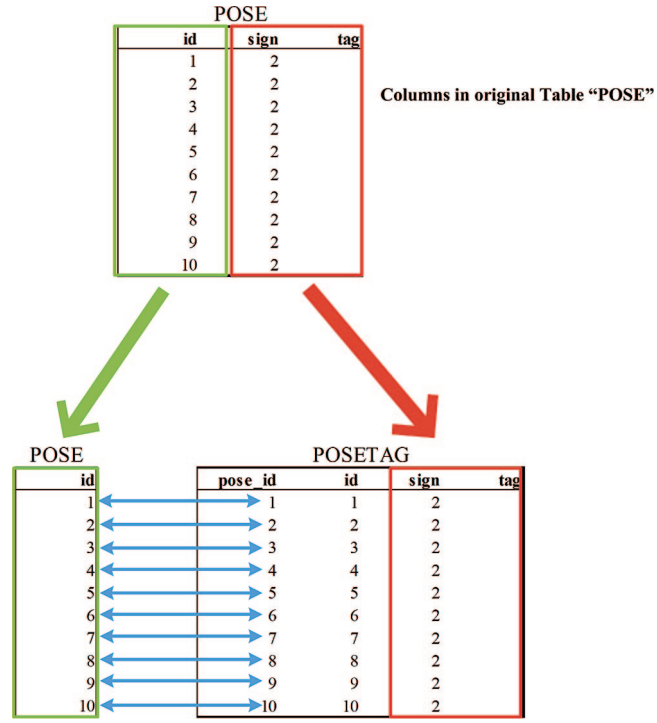

Figure 4. Vertical partitioning of Table "POSE". The value of Column "posetag.id" in "POSETAG" (Table 4) has no relation to that of Column "pose_id". The value of Column "sign" can be "1", "2", "3", and "6", which mean "in process", "normally finished", "finished with errors", and "unhandled", respectively.

This relation is analogous to foreign key constraints in the context of relational databases, even though MyISAM[30] (see Storage Engine) still does not support it in the latest version (MySQL 6.0). In this way, when handling the status of poses, the database management system only scanned the smaller table ("POSETAG") with the fixed row format instead of the large and dynamic table ("POSE"). In addition, different storage engines could be applied on different tables (see **Storage Engine**).

Storage Engine. Most tables in the database are constructed with MyISAM,[30] which is the default storage engine of MySQL due to its very low overhead, except for Table "POSETAG" constructed with InnoDB.[31] InnoDB uses more memory as cache to achieve a high performance. In fact, the database engine does not allow parallel accesses: a client has to lock the object for an update to prevent conflicting with other clients. InnoDB implements row-level locking, so that InnoDB only locks the rows needed for update instead of locking the entire table as MyISAM does. This feature is advantageous to concurrent updating from multiconnections with low lock wait ratio (LWR). The LWR is the percentage of queries that are required to wait for object locks to be released so that the query can itself acquire a lock on the object, e.g., many clients can update statuses of poses by modifying the Table "POSETAG" synchronously. The parallel performance of DVSDMS can be estimated by the LWR. A low LWR means that the performance loss due to the multiple connections of database is marginal. By using InnoDB instead of MyISAM as the storage engine of "POSETAG", the LWR of the database is reduced significantly, specifically in the ranking process MyISAM often induced a deadlock (LWR≈100%) while InnoDB reduced LWR to less than 0.1%.

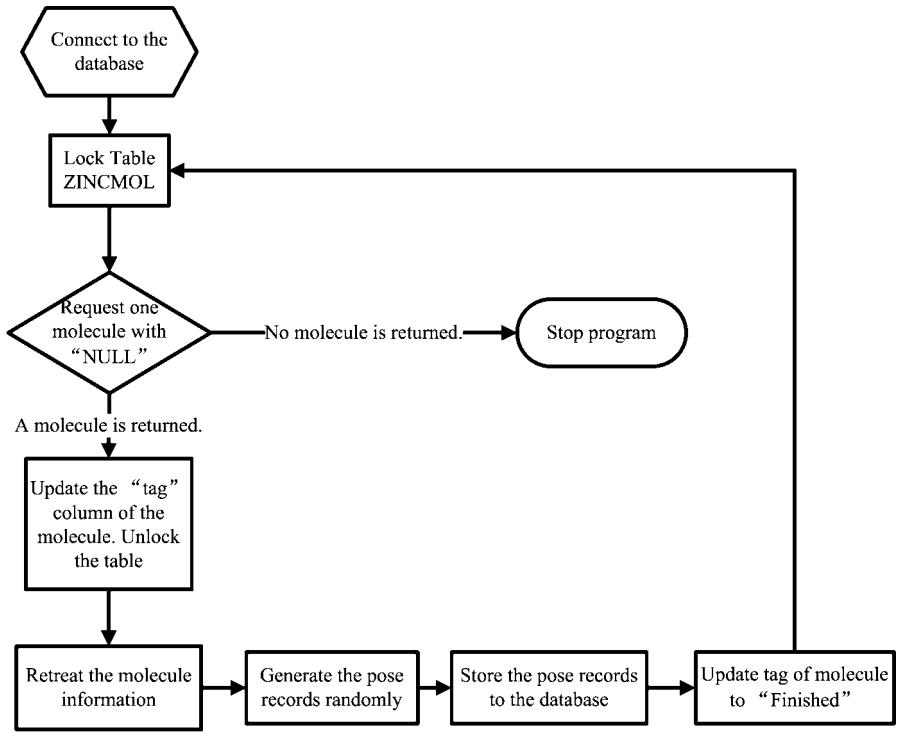

Figure 5. The flowchart of the docking emulator. In the benchmark the average amount of poses for each molecule is 37 ± 3 and the average size of each pose is 1 ± 0.5KB, which are consistent to the average in the real application to identify kinase inhibitors.

Local Cache and Bulk Update. The performance of the database can be improved by using local cache and bulk update, which were especially important for the short-term process, such as ranking in our calculation. Otherwise, the applications communicated with the database with high frequency and cast a heavy burden on the database. In our application, 2000 poses were retrieved from the database by a single SQL command and stored in the local memory. During the calculation, every result of single pose was stored in memory temporarily, which works as local cache. After all the ranking calculations of these poses had been finished, the client sent the results to the database and updated multiple rows (bulk update) by another SQL command. In this way, the clients only need to communicate with the database twice for handling 2000 poses. Note that if the amount of poses retrieved by the client using a single SQL command is too small, there will be no obvious increment of performance. Conversely, a large amount of poses will increase the individual query time and the memory use of the database server.

Compressing Molecule Files. Compressing molecule files is favorable for saving disk space and network bandwidth and decreasing the I/O intensity. For a normal PDB or MOL2 file, compression reduced its size by about 75% using zlib.[32] The compression and decompression work was performed by the clients and would not influence the load of the database server. After compression, the Table "POSE" used 93.1GB hard disk for about 100.8 million poses and their energies (Table 2).

RESULTS

Benchmark. Since we could not access a large computer cluster, we wrote an emulator of a docking client to test the performance of DVSDMS under conditions typical of very large clusters where the bottleneck is the master rather than the clients. In fact, our emulator does not run the real docking calculation but only requests jobs from the master and returns the output data, such as randomized interaction energies and efficiencies as well as binding poses (Figure 5). Note that emulated docking does not require any CPU time, which is essential to estimate the maximal rate of distribution of molecules (maxRoD) to the computer clusters we could access (≤200 nodes). The benchmark database server was built on a dedicated desktop computer, which had two Xeon 3.06 GHz CPUs, 2 GB of RAM, and a 250GB normal speed hard disk drive (maximal reading speed 50MB/s). The efficiency of a parallel docking program based on the master-worker scheme can be measured by the maxRoD of the master. A series of test cases was performed with the amount of emulators ranging from 1 to 160 and running one emulator per core. Two Beowulf clusters at the University of Zürich were used: The Etna and Matterhorn, which are separated from the master by one and four switches, respectively. Both clusters and the master are on a Gigabit network. The time required for emulated docking of 128000 molecules, which is a subset of ZINC library, is shown in Figure 6 for different numbers of emulators. The minimal times for the simulated docking are 1816 and 2292 s on the Etna and Matterhorn cluster, respectively. The lower amount of switches to reach Etna yields about 26.2% performance improvement in the benchmark. Note that the system load of the master does not hit 100% when the maxRoD reaches the plateau at about 12 cores (or emulators), which indicates that the network delay rather than the capacity of the master limits the overall performance. The maxRoD of DVSDMS running on Etna and Matterhorn are 70 and 56, respectively. The MaxRoD of the MPI version of DOCK[14] is about 19 (see the Supporting Information). Therefore the DVSDMS is about two times faster than the MPI version of DOCK on the Blue Gene/L platform. Even though the HTC version of DOCK

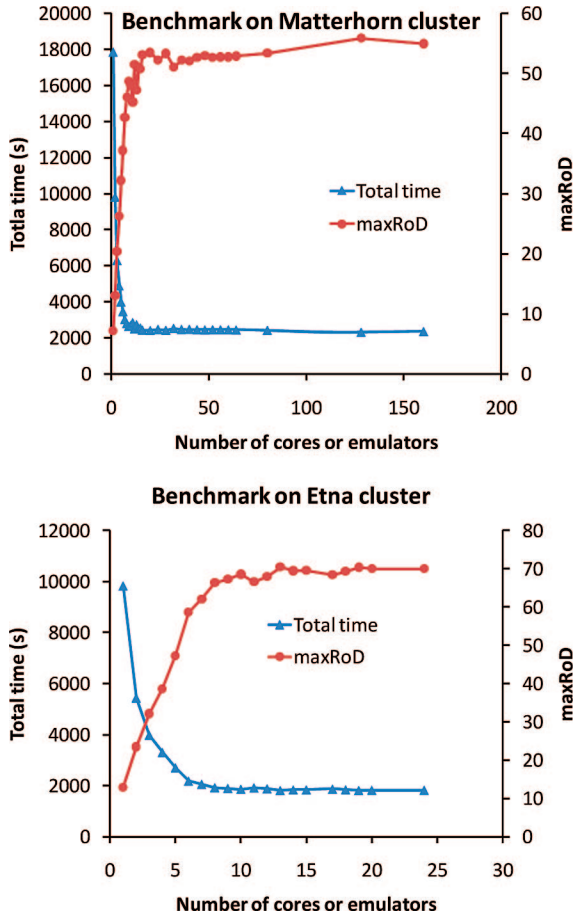

Figure 6. The duration of simulated docking of 128000 molecules with different numbers of emulated docking clients. Note that each core runs only one emulator. (Top) The emulators were submitted to the Matterhorn cluster (80 compute nodes each with dual-processor Opteron 2.4 GHz or 2.6 GHz), from which data needed to pass 4 switches to reach the database server. With more than 12 emulators, the database server, i.e., the master of DVSDMS, achieves a maximal rate of distribution of molecules (maxRoD) of about 56 per second on average (128000/2292s). (Bottom) On the Etna cluster (3 compute nodes each with dual Quad Core Xeon 2.33 GHz), the data only needed to pass 1 switch to reach the database server, and the maxRoD increased to about 70 per second on average (128000/1816s). Note that the delay due to the network equipment is negligible when there is a small quantity of requests. In both benchmarks however, the database server needs to respond to thousands of requests per second, so that the network delay limits the overall performance, and the plateau of the maxRoD is due to the network.

and the DVSDMS have similar capacities for distributing molecules (up to at least 16384 processors) it is easier to implement other docking engines in DVSDMS than to write a specific HTC version for each docking engine.

Performance in Production. The database server, backup machine, and the user's machine (Figure 2) were all built on the Linux desktop PC of the first author, which had dual-core 3.4 GHz Pentium D, 3 GB of RAM, and 3×320GB hard disks. The disk could read data at about 70 MB/s.

About 1.5 million compounds out of 3.8 millions in the ZINC library passed the filters used for eliminating the non-drug-like molecules (molecular weight <500 Da, number of rotatable bonds ≤ 7, number of hydrogen bond donors ≥ 1, and number of hydrogen bond acceptors ≥ 1). For the compounds which did not pass the filters the column "zincmol.tag" was set to "not passed". About 15 to 20 min were required for docking a single compound into the receptor and CHARMM minimization of the poses (with a rigid protein) on an Opteron Processor 252 (2.6 GHz). After all the docking jobs had finished, about 100 million poses were stored in Table "POSE", 80% of which were selected for ranking according to their interaction energies and efficiency. The computational time required for ranking a pose ranged from 1 s to 15 min depending on whether further calculations were needed for the pose and/or the convergence of quantum mechanical calculations.

The jobs were carried out by the Etna and Matterhorn clusters and some standalone computers simultaneously. The number of processors varied dynamically depending on the availabilities. The load of the database was low during the docking process using about 500 clients (<100 queries/s, about 100KB/s traffic of network, and $<10\%$ overall system load), because each docking client only communicated with the database 3 to 5 times per hour. Therefore, if the master of DVSDMS can distribute 60 molecules per second and a client requests 5 molecules per hour, the database server can support up to 43200 computing clients with nearly linear scalability if the clients are well synchronized. In contrast, during the ranking process, the load of the database was high (about 300 queries/s, 500−1000 KB/s traffic of network, and about 30% overall system load), because ranking 2000 poses usually took less than 5 min, and each of the 500 clients communicated with the database more than 12 times per hour. Note that docking and ranking could be combined sequentially to reduce the load of database.

CONCLUSIONS

DVSDMS uses freely available database software for efficient and automatic virtual screening distributed on Linux platforms. The essential concept of DVSDMS is the separation of data management from the main jobs in virtual screening, i.e., docking and ranking. In this way, the user has full flexibility on the choice of software for docking and ranking as well as hardware. Organized by DVSDMS, jobs are dispatched to each computing client to optimally exploit the available resources even in the case of heterogeneous hardware. Users not only can control and inspect the computing process but also attain consistent and logically organized data while computing is in progress or finished.

Because docking and ranking consist of many independent jobs, they are typically suited for a coarse-grained parallel architecture. In DVSDMS, computing clients do not communicate among each other but only with the database server. When a job is requested by a client, the database server scans the handling statuses and returns an unhandled job. In this way, a priori job partitioning is not required, and the overall computational load can be distributed equally to all computing clients. Moreover, the queue of jobs can be modified at any time; new jobs can be added to the computational pipeline by appending them to the database, while existing jobs can be removed before they start to run.

Upon performance tuning, the evaluation of the number of queries, the duration of queries, and the data flow indicate that the overall performance of DVSDMS is good. In particular, local cache and bulk updates reduce the query number; database index, proper storage engine, and database partitioning speed up queries; and data compression reduces

the data flow. Furthermore, since the database management system works as the master of DVSDMS, most of its sophisticated techniques (e.g., read write splitting and database cluster) can be applied directly to improve the performance of the master avoiding the overload without modifying the code for docking and ranking. In a docking benchmark, the master of DVSDMS built on a low cost Linux PC could distribute about 60 molecules per second. Furthermore nearly linear scalability of DVSDMS is expected up to 50000 nodes.

In the application presented here, docking the ZINC library into the receptor tyrosine kinase EphB4 with AutoDock and ranking poses under the control of DVSDMS, a low cost Linux PC was perfectly competent for the database server connected to about 500 computing clients. Since, the database management system MySQL and the program language Python are both open-source projects, DVSDMS can be applied in high-throughput virtual screening campaigns without restrictions typical of proprietary software.

AVAILABILITY OF DVSDMS

All scripts are available at http://biocroma.uzh.ch/zhou/dvsdms/.

ACKNOWLEDGMENT

We thank Armin Widmer for the continuous and extremely helpful support for the molecular modeling program Witnotp, and Philipp Schütz for useful suggestions and comments to the manuscript. This work was supported by a Swiss National Science Foundation grant to A.C. Most of the calculations were performed on the Etna and Matterhorn Beowulf cluster at the University of Zürich.

Supporting Information Available: Docking details, and the estimation of the maximum amount of molecules distributed per seconds for the MPI version of dock. This material is available free of charge via the Internet at http://pubs.acs.org.

REFERENCES AND NOTES

(1) Vangrevelinghe, E.; Zimmermann, K.; Schoepfer, J.; Portmann, R.; Fabbro, D.; Furet, P. Discovery of a Potent and Selective Protein Kinase CK2 Inhibitor by High-Throughput Docking. *J. Med. Chem.* **2003**, *46*, 2656–2662.
(2) Jorgensen, W. L. The Many Roles of Computation in Drug Discovery. *Science* **2004**, *303*, 1813–1818.
(3) Desai, P. V.; Patny, A.; Gut, J.; Rosenthal, P. J.; Tekwani, B.; Srivastava, A.; Avery, M. Identification of Novel Parasitic Cysteine Protease Inhibitors by Use of Virtual Screening. 2. The Available Chemical Directory. *J. Med. Chem.* **2006**, *49*, 1576–1584.
(4) Huang, D. Z.; Luthi, U.; Kolb, P.; Cecchini, M.; Barberis, A.; Caflisch, A. In Silico Discovery of Beta-Secretase Inhibitors. *J. Am. Chem. Soc.* **2006**, *128*, 5436–5443.
(5) Brown, S. P.; Muchmore, S. W. High-Throughput Calculation of Protein-Ligand Binding Affinities: Modification and Adaptation of the MM-PBSA Protocol to Enterprise Grid Computing. *J. Chem. Inf. Model.* **2006**, *46*, 999–1005.
(6) Kolb, P.; Huang, D.; Dey, F.; Caflisch, A. Discovery of Kinase Inhibitors by High-Throughput Docking and Scoring Based on a Transferable Linear Interaction Energy Model. *J. Med. Chem.* **2008**, *51*, 1179–1188.
(7) Kolb, P.; Kipouros, C. B.; Huang, D.; Caflisch, A. Structure-Based Tailoring of Compound Libraries for High-Throughput Screening: Discovery of Novel EphB4 Kinase Inhibitors. *Proteins* **2008**, *73*, 11–18.
(8) Friesner, R. A.; Banks, J. L.; Murphy, R. B.; Halgren, T. A.; Klicic, J. J.; Mainz, D. T.; Repasky, M. P.; Knoll, E. H.; Shelley, M.; Perry, J. K.; Shaw, D. E.; Francis, P.; Shenkin, P. S. Glide: A New Approach for Rapid, Accurate Docking and Scoring. 1. Method and Assessment of Docking Accuracy. *J. Med. Chem.* **2004**, *47*, 1739–1749.
(9) Halgren, T. A.; Murphy, R. B.; Friesner, R. A.; Beard, H. S.; Frye, L. L.; Pollard, W. T.; Banks, J. L. Glide: A New Approach for Rapid, Accurate Docking and Scoring. 2. Enrichment Factors in Database Screening. *J. Med. Chem.* **2004**, *47*, 1750–1759.
(10) Goodsell, D. S.; Olson, A. J. Automated Docking of Substrates to Proteins by Simulated Annealing. *Proteins-Structure Function and Genetics* **1990**, *8*, 195–202.
(11) Zhang, S. X.; Kumar, K.; Jiang, X. H.; Wallqvist, A.; Reifman, J. DOVIS: An Implementation for High-Throughput Virtual Screening Using AutoDock. *Bmc Bioinformatics* **2008**, 9.
(12) Kuntz, I. D.; Blaney, J. M.; Oatley, S. J.; Langridge, R.; Ferrin, T. E. A Geometric Approach To Macromolecule-Ligand Interactions. *J. Mol. Biol.* **1982**, *161*, 269–288.
(13) Moustakas, D. T.; Lang, P. T.; Pegg, S.; Pettersen, E.; Kuntz, I. D.; Brooijmans, N.; Rizzo, R. C. Development and Validation of a Modular, Extensible Docking Program: DOCK 5. *J. Comput.-Aided Mol. Des.* **2006**, *20*, 601–619.
(14) Peters, A.; Lundberg, M. E.; Lang, P. T.; Sosa, C. P. High Throughput Computing Validation for Drug Discovery Using the DOCK Program on a Massively Parallel System. *RedPaper* **2008**, *REDP-4410−00*.
(15) Mullen-Schultz, G. L.; Sosa, C. P. IBM System Blue Gene Solution. *Application Development.* **2007**, SG24−7179.
(16) Kolb, P.; Caflisch, A. Automatic and Efficient Decomposition of Two-Dimensional Structures of Small Molecules for Fragment-Based High-Throughput Docking. *J. Med. Chem.* **2006**, *49*, 7384–7392.
(17) Widmer, A., WITNOTP: A Computer Program for Molecular Modeling. Novartis: Basel, 1997.
(18) Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. Charmm - a Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *J. Comput. Chem.* **1983**, *4*, 187–217.
(19) Stewart, J. J. P. Optimization of Parameters for Semiempirical Methods I. Method. *J. Comput. Chem.* **1989**, *10*, 209–220.
(20) Huang, D.; Caflisch, A. Efficient Evaluation of Binding Free Energy Using Continuum Electrostatics Solvation. *J. Med. Chem.* **2004**, *47*, 5791–5797.
(21) Zhou, T.; Huang, D.; Caflisch, A. Is Quantum Mechanics Necessary for Predicting Binding Free Energy. *J. Med. Chem.* **2008**, *51*, 4280–4288.
(22) Charifson, P. S.; Corkery, J. J.; Murcko, M. A.; Walters, W. P. Consensus scoring: A Method for Obtaining Improved Hit Rates from Docking Databases of Three-dimensional Structures into Proteins. *J. Med. Chem.* **1999**, *42*, 5100–5109.
(23) Clark, R. D.; Strizhev, A.; Leonard, J. M.; Blake, J. F.; Matthew, J. B. Consensus Scoring for Ligand/protein Interactions. *J. Mol. Graphics Modell.* **2002**, *20*, 281–295.
(24) Gohlke, H.; Klebe, G. Statistical Potentials and Scoring Functions Applied to Protein-ligand Binding. *Curr. Opin. Struct. Biol.* **2001**, *11*, 231–235.
(25) Irwin, J. J.; Shoichet, B. K. ZINC - A Free Database of Commercially Available Compounds for Virtual Screening. *J. Chem. Inf. Model.* **2005**, *45*, 177–182.
(26) Momany, F. A.; Rone, R. Validation of the General-Purpose Quanta(R)3.2/Charmm(R) Force-Field. *J. Comput. Chem.* **1992**, *13*, 888–900.
(27) *SQLAlchemy*, 0.4.6; Michael Bayer: New York, NY, 2008.
(28) LaCour, J.; Haus, D.; Menten, d. G. Elixir. http://elixir.ematia.de/trac/ (accessed 31, Oct, 2008).
(29) MySQL::MySQL 5.1 Reference Manual::17.2.3 HASH Partitioning. http://dev.mysql.com/doc/refman/5.1/en/partitioning-hash.html (accessed 31, Oct, 2008).
(30) MySQL::MySQL 5.0 Reference Manual::13.1 The MyISAM Storage Engine. http://dev.mysql.com/doc/mysql/en/myisam-storage-engine.html (accessed 31, Oct, 2008).
(31) InnoDB Website. http://www.innodb.com/ (accessed 31, Oct, 2008).
(32) Gailly, J.; Adler, M. zlib Home Site. http://www.zlib.net/ (accessed 31, Oct, 2008).
(33) MySQL::MySQL 5.1 Reference Manual::10.5 Data Type Storage Requirements. http://dev.mysql.com/doc/refman/5.1/en/storage-requirements.html (accessed 31, Oct, 2008).
(34) ADT/AutoDockTools-AutoDock. http://autodock.scripps.edu/resources/adt/index_html (accessed 31, Oct, 2008).
(35) MySQL::The world's most popular open source database. http://www.mysql.com/ (accessed 31, Oct, 2008).

CI800295Q