BIOINFORMATICS

Vol. 00 no. 00 2015
Pages 1–7

A Distributed Computing Platform for Large-Scale Computational Protein Design

Yuchao Pan,[1] Yuxi Dong,[2] Jianyang Zeng[3,*] and Wei Xu[4]

[1], [2], [3] and [4]

ABSTRACT

Motivation:

Results:

Availability: Our software is available and distributed open-source under the BSD license on the date that the paper is published. We will also provide a Amazon EC2 AMI virtual machine to allow people use our system in the public cloud.

Contact: zengjy321@tsinghua.edu.cn

1 INTRODUCTION

Protein design is an important method in drug discovery, (XUW: what else??) and other life science research areas. Due to the long cycle and high cost in wet-lab experiments, Computational Protein Design (CPD) has become an important tool for protein engineering (Alvizo et al., 2007). People have successfully demonstrated the effectiveness of CPD in applications such as peptide synthesis (Ottl et al., 1996), protein-protein interactions (Roberts et al., 2012), artificial gene synthesis (Villalobos et al., 2006).

(XUW: what is the difference/relation between CPD and SCPR?) Specifically, in the structure-based computational protein design (SCPR) problem, the goal is to predict amino acid sequences that will fold to a specific protein structure. More precisely, the aim for CPD is to find the global minimum energy conformation (GMEC) based on the desired energy function.

The protein design problem has been proven NP-hard (Pierce and Winfree, 2002). On approach is to obtain approximate solutions, providing no guarantees on the optimal solution. This problem is modeled as a MAP-MRF inference problem (Yanover et al., 2006), which can be approximated by a Linear Programming Relaxation (LPR) problem (Wainwright et al., 2005).

On the other hand, there are several methods which can solve the GMEC problem exactly. Of course, to solve an NP-hard problem exactly, scaling to a larger protein is the major concern of current these algorithms.

There are two considerations to scale to a larger protein. The first is to redesign the algorithms to limit the search space as much as possible. Such as DEE/A* (OSPREY, Donald Lab at DUKE University) (XUW: used XXX technique to reduce the search space, while .. (summarize their key ideas)) Branch-and-Bound Search (Hong and Lozano-Pérez, 2006), tree decomposition (Xu and Berger, 2006), AND/OR Branch-and-Bound (Marinescu and Dechter, 2009), Integer Linear Programming (Kingsford et al., 2005), Cost Network Function (Traoré et al., 2013).

The second way is to use more machines to perform the computation and tries to achieve parallel performance. As the search space can be partitioned to allow each node to handle one partition, we can speed up the search using more machines. With the development of cloud computing, the cost of computation resources is going down rapidly, making distributed brutal force solutions more plausible. Folding@Home is an early attempt to cultivate unused cycles on desktop PCs to perform protein computation. However, by the definition of NP-hard, massive computation resources are wasted even for a moderate-sized problem.

It is difficult to combine the optimized algorithms in a large distributed computing infrastructure. The main reason is that the optimization part of these algorithms often requires global states, such as the upper and lower bounds in the branch-and-bound (BnB) algorithm. Unfortunately, it is a known hard problem to efficiently sharing such a state in a "cloud" system built with commodity hardware and networking, especially in the presence of node failures.

As the first work to combine a highly optimized GMEC algorithm with massively scalable cloud-based system, we designed a version of the branch-and-bound (BnB) search algorithm over a customized Hadoop framework. We demonstrated that the system scales near-linearly to hundreds of computation servers, attempting hundreds of billions of braches in parallel, while still takes advantage of all optimizations in the algorithm.

In our method, the DEE criteria(XUW: need to explain this?) is applied to prune the infeasible rotamers not only as a pre-filtering algorithm but also in the branch step. Since the efficiency of the branch-and-bound searching algorithm heavily depends on the tightness of the bound, we use message passing algorithm for approximating (Globerson and Jaakkola, 2008) and mini-bucket elimination (Rollon and Dechter, 2010) to compute the lower bound, and use simulated annealing to find a relatively better solution as our upper bound. In distributed environment, each node independently discovers the bound in its part of the search space, and it is essential to quickly share the global bound to each node. The global bound is our global state for algorithm optimization.

Our key observation is that the global bound does not affect the correctness of the algorithm: if a node does not receive the global

[*] To whom correspondence should be addressed.

state, it just uses a sub-optimal bound and result is no more than performing some unnecessary computation. Thus we can share the bound in a best-effort way and tolerate the occasional inconsistency of the states. As we will show in the paper, the extra overhead is negligible and the gain of performance is significant comparing to the brutal force approach.

In this paper, we demonstrate two approaches to perform the best-effort state sharing. The first approach uses probabilistic data propagation, resulting in an algorithm that runs on unmodified MapReduce () framework. The second approach is based on asynchronous communication and eventual consistency model (), which requires minor modifications of the Hadoop framework, but can improve performance by XX

Our system inherits all fault-tolerance benefits of the cloud-based systems. In one of our experiments, we killed XXX computation nodes and the analysis still completed normally. This feature is the key to make the cloud economics model work: researchers can take advantage of the off-peak machine time to perform the GMEC tasks and can kill some tasks whenever there is better use of the machines the killed tasks will automatically restart on other machines without affecting the final results. The cost of off-peak machine times (such as the spot instances in Amazons EC2 cloud computing platform () http://aws.amazon.com/ec2/purchasing-options/spot-instances/) is only a fraction of the normal price, making our approach financially practical.

We have three contributions in this paper

1. We integrated several optimizations to improve the BnB algorithm. (XUW: key ideas of the algorithms)
2. We demonstrated that by relaxing the consistency requirements of the global bound in the BnB algorithm, we create a massively scalable computation system that runs the highly optimized version of BnB algorithm to solve GMEC problems in protein design. The system runs efficiently on commodity cloud computing environment and tolerates failures effectively.
3. We obtain the optimal results on XX protein design problems, which was not possible using even the state-of-the-art single machine algorithms. (XUW: more comp-bio contributions here?)

We want to emphasis that although this paper focuses on branch-and-bound algorithm, our methodology of handling global states by relaxing the consistency requirements can be applied to many different search algorithms of the same nature.

The remaining of the paper goes as follows..

2 METHODS

2.1 Background

2.1.1 *Problem formulation*

The protein design problem can be described as a pairwise markov networks. All the residues of the protein can be regarded as the vertices of the network, and the interaction between residues is the edges. Let $G = (V, E)$ be the markov network of the protein design problem, then for each vertex $v \in V$, there is an available states set X_v, which is the rotamer set of the residue position v in the protein chain. The aim of the CPD problem is to find the optimal assignment $\mathbf{x}^*$ that minimizes the total energy, namely

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \left(\sum_{v \in V} \theta_v(x_v) + \sum_{(u,v) \in E} \theta_{uv}(x_u, x_v) \right)$$

where $x_v \in X_v$ is the rotamer at residue position v, $\theta_v(x_v)$ is the potential energy between the internal atoms of the rotamer, and $\theta_{uv}(x_u, x_v)$ is the potential energy between rotamer x_u at residue position u and x_v.

In order to find the optimal solution of a NP-hard problem, we often need to search over a huge state space. As many popular methods to solve the CPD problem, firstly use dead-end elimination as a pre-filtering algorithm to prune the rotamers that are not part of the GMEC, and thus reduce the search space. Then we use branch-and-bound algorithm on a distributed computing platform to search the remain search space.

2.1.2 *Branch-and-Bound*

Branch-and-Bound (BnB) is a widely-used search algorithm for solving various combinatorial optimization problems. This algorithm constantly divides the state space into several smaller sub-spaces (this step is called *branching*) and then calculate the bound for each sub-spaces (this step is called *bounding*). After that, those sub-spaces which certainly not contain the optimal solution (i.e. the lower bound is larger than the known upper bound) are discarded.

To be more specific, suppose we want to solve an optimization problem through minimizing an energy function over a state space S. The algorithm has two main steps:

Branching: In this step, the state space S is split into two or more sub-space $S_1, S_2, \ldots, S_m$ such that $S_1 \cup S_2 \cup \cdots \cup S_m = S$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$.

Bounding: In this step, we compute the lower bound and upper bound of each sub-space S_i, denote by $LB(S_i)$ and $UB(S_i)$. Let $GUB = \min_{1 \leq i \leq m} UB(S_i)$ be the minimum upper bound, then we can prune the sub-space S_i if $LB(S_i) > GUB$, since there exists an element that is better than all elements of S_i.

The aforementioned combination of *branching* and *bounding* steps is recursively performed until the state space only contains a single element.

Assume we the network contains n vertices which are numbered from 1 to n, and let X_i denote the state set of vertex i. Here is the pseudo code of Branch-and-Bound algorithm.

Here Q is usually a FIFO or priority queue, and we maintain a global variable GUB to store the best solution. An efficient lower bound of a state space will be proposed in Section 2.2.2, and upper bound in Section 2.2.3.

2.1.3 *Local Dead-End Elimination Algorithm*

Algorithm 1 Branch-and-Bound Algorithm

```
function BRANCHANDBOUND($G = (V, E), X, \Theta$)
    $S_V \leftarrow X_1 \times X_2 \times \cdots \times X_n$
    $GUB \leftarrow$ UPPERBOUND($S_V$)
    ADD($Q, S_V$)
    while $Q$ is not empty do
        $S \leftarrow$ NEXTELEMENT($Q$)
        if LOWERBOUND($S$) $\geq GUB$ then
            continue
        end if
        $(S_1, S_2, ..., S_m) \leftarrow$ BRANCH($S$)
        for $i \leftarrow 1$ to $m$ do
            $GUB \leftarrow \min(GUB,$ UPPERBOUND($S_i$))
        end for
        for $i \leftarrow 1$ to $m$ do
            if LOWERBOUND($S_i$) $< GUB$ then
                ADD($Q, S_i$)
            end if
        end for
    end while
    return $GUB$
end function
```

The dead-end elimination (DEE) algorithm is an efficient method to eliminate infeasible variable states. For a variable x_v, and two variable states x_v^i and x_v^j in X_v, if the following condition is satisfied, then state x_v^i can be eliminated, which reduces the search space.

$$\begin{aligned}&\theta_v(x_v^i) + \sum_{(u,v)\in E} \min_{x_u \in X_u} \theta_{uv}(x_u, x_v^i)\\ &> \theta_v(x_v^j) + \sum_{(u,v)\in E} \max_{x_u \in X_u} \theta_{uv}(x_u, x_v^j)\end{aligned} \tag{1}$$

The more powerful criterion that improved by Goldstein (1994) is

$$\theta_v(x_v^i) - \theta_v(x_v^j) + \sum_{(u,v)\in E} \min_{x_u \in X_u} [\theta_{uv}(x_u, x_v^i) - \theta_{uv}(x_u, x_v^j)] > 0 \tag{2}$$

We apply the Goldstein DEE criterion in (2) to the function BRANCH. Let $D(X)$ be the set of variables that have been searched, and $U(X) = V \setminus D(X)$ be the set of variables which has not been determined yet. Consider two variable states x_v^i and x_v^j in an undetermined variable x_v, the Goldstein DEE criterion we use in the BRANCH functions is

$$\begin{aligned}&\theta_v(x_v^i) - \theta_v(x_v^j) + \sum_{\substack{(u,v)\in E\\ u\in D(X)}} [\theta_{uv}(x_u, x_v^i) - \theta_{uv}(x_u, x_v^j)]\\ &+ \sum_{\substack{(u,v)\in E\\ u\in U(X)}} \min_{x_u \in X_u} [\theta_{uv}(x_u, x_v^i) - \theta_{uv}(x_u, x_v^j)] > 0\end{aligned} \tag{3}$$

By applying the DEE criterion in Eq. (3), we can eliminate a large number of infeasible variable states, and thus significantly reduces the branch space.

Here, the DEE criterion is incorporated into each branch step. To distinguish it from most other DEE criteria that are applied before search algorithms (e.g. A*(Gainza et al., 2013)), we call the criterion in Eq. (3) integrated into the branch step the *local DEE criterion.*

2.1.4 Lower Bound

Naive Lower Bound. A naive lower bound of the energy function in protein design can be easily computed by considering the best possible rotamer assignment in each residue, which is

$$\sum_{v\in V} \min_{x_v \in X_v} \left(\theta_v(x_v) + \sum_{\substack{(u,v)\in E\\ u<v}} \min_{x_u \in X_u} \theta_{uv}(x_u, x_v) \right) \tag{4}$$

That is, the naive lower bound of the current state space X can be written as

$$\begin{aligned}LB_1(X) =& g(X) + \sum_{v\in U(x)} \min_{x_v \in X_v} \left(\theta_v(x_v) + \sum_{u\in D(X)} \theta_{uv}(x_u, x_v) \right.\\ &\left. + \sum_{\substack{u\in U(X)\\ u<v}} \min_{x_u \in X_u} \theta_{uv}(x_u, x_v) \right)\end{aligned} \tag{5}$$

where we leave $(u, v) \in E$ out from the summation notation for simplifying the expression, and $g(X)$ is the energy of the determined variables (i.e. those residues in which the rotamers have been determined), that is

$$g(X) = \sum_{v\in D(X)} \theta_v(x_v) + \sum_{\substack{u,v\in D(X)\\ u<v}} \theta_{uv}(x_u, x_v)$$

Efficient Lower Bound. By observing the formula of the naive lower bound in Eq. (4), we see that every edge-energy function is only used for one vertex (i.e. the vertex has greater index in Eq. (4)). If we split θ_{uv} into two functions β_{uv} and β_{vu} where $\beta_{uv}(x_u, x_v) + \beta_{vu}(x_v, x_u) = \theta_{uv}(x_u, x_v)$ for all x_u, x_v, then the formula of (4) becomes

$$\begin{aligned}\max &\sum_{v\in V} \min_{x_v \in X_v} \left(\theta_v(x_v) + \sum_{(u,v)\in E} \min_{x_u \in X_u} \beta_{uv}(x_u, x_v) \right)\\ s.t.\ &\beta_{uv}(x_u, x_v) + \beta_{vu}(x_v, x_u) = \theta_{vu}(x_u, x_v)\\ &\forall (u, v) \in E, x_u \in X_u, x_v \in X_v\end{aligned} \tag{6}$$

The above optimization problem is a convex dual of MAPLPR, which can be solved by Convergent Message Passing Algorithms (Globerson and Jaakkola, 2008).

Let β^* be the functions that found by message passing algorithm, then the lower bound of the current state space X becomes

$$LB_2(X) = g(X) + \sum_{v \in U(X)} \min_{x_v \in X_v} \left(\theta_v(x_v) + \sum_{u \in D(X)} \theta_{uv}(x_u, x_v) \right. \\ \left. + \sum_{u \in U(X)} \min_{x_u \in X_u} \beta^*_{uv}(x_u, x_v) \right) \quad (7)$$

Since β^* may not be the best functions for any state space, it is necessary because we can not compute it for all the searched state space.

If we compute LB_2 directly, then the time complexity is $O(n^2m^2)$, where n is the number of mutable residues, and m is the number of rotamers per residue. If we firstly compute a table p that $p_{uv}(x_v) = \min_{x_u \in X_u} \beta^*_{uv}(x_u, x_v)$ with time $O(n^2m^2)$ and space $O(n^2m)$, then the time of computing LB_2 decrease to $O(n^2m)$.

Mini-Bucket. The mini-bucket elimination (MBE) is a well-known approximation algorithm for graphical models (Dechter and Rish, 2003; Rollon and Dechter, 2010; Rollon and Larrosa, 2006), and it gives a bound when the induced width of the graph is too large. The idea of mini-bucket elimination is to eliminate variables, and the time and space complexity of MBE is $O(m^i)$ where i is a user controlled parameter that restrict the size of the scopes of each functions. We also apply this algorithm in our method to

2.2 Upper Bound

Upper bound is different from lower bound, we often use a relatively better solution in X as its upper bound. There are many meta-heuristic methods have been applied to it, such as Monte-Carlo with simulated annealing (Kuhlman and Baker, 2000; Voigt et al., 2000), and genetic algorithms (Raha et al., 2000). These approaches can usually find a relatively better solution quickly but without any guarantees of accuracy. Thus, these methods provide us with an efficient upper bound.

In our method, we choose simulated annealing as our upper bound algorithm. Simulated annealing (SA) is a generic probabilistic meta-heuristic method for the global optimization problem, and it is often used when the search space is discrete. The SA heuristic is started with an arbitrary initial state. At each step, consider a neighbouring state s' of the current state s, and probabilistically decides between moving to s' or staying in s. The probabilities ultimately lead the system to the states with lower energy.

The initial state $\mathbf{x}^0$ of our SA heuristic is based on the lower bound function LB_2, which is

$$\mathbf{x}^0_v = \arg\min_{x_v \in X_v} \left(\theta_v(x_v) + \sum_{u \in D(X)} \theta_{uv}(x_u, x_v) + \sum_{u \in U(X)} p_{uv}(x_v) \right)$$

Let $\mathbf{x}^S$ be the best solution found by Simulated Annealing Algorithm, then

$$UB(X) = g(\mathbf{x}^S)$$

The time complexity of the SA heuristic T_{SA} is based on the number of iteration rounds I and the time of calculating the energy function T_{EF}, that is $T_{SA} = I * T_{EF}$. Consider the current state $\mathbf{x}$ and a neighbouring state $\mathbf{x}'$, there is only one rotamer at some residue position is different, namely $\mathbf{x} = (x_1, x_2, ..., x_i, ..., x_n)$ and $\mathbf{x}' = (x_1, x_2, ..., x'_i, ..., x_n)$ at some position i. We have already known the energy $g(\mathbf{x})$ of $\mathbf{x}$, and now we can compute the energy of the neighboring state $\mathbf{x}'$ as follows.

$$g(\mathbf{x}') = g(\mathbf{x}) - \left(\theta_i(x_i) + \sum_{j \neq i} \theta_{ji}(x_j, x_i) \right) \\ + \left(\theta_i(x'_i) + \sum_{j \neq i} \theta_{ji}(x_j, x'_i) \right)$$

Thus the time T_{EF} is optimized to $O(n)$ where n is the number of mutable residues, and then $T_{SA} = O(n^2 + I * n)$ where n^2 is the time of calculating the energy of the initial state.

3 DISTRIBUTED BRANCH-AND-BOUND

3.1 Baseline System: Branch-and-Bound on MapReduce

3.1.1 MapReduce

MapReduce (Dean and Ghemawat, 2008) is a programming model proposed by Google which is used for massive parallel and distributed computing on large data sets. The computation takes a set of $(key, value)$ pairs as input, and produces a set of output $(key, value)$ pairs. It is composed of the following two main steps:

Map: The input data set is divided into several parts, and each part is processed by a worker to produce a set of intermediate $(key, value)$ pairs in parallel. The MapReduce library groups together the values with the same key, and then passes them to the REDUCE function.

Reduce: The REDUCE function accepts an intermediate key, and a set of values for that key. Each REDUCE function is also processed in parallel, and produces a set of $(key, value)$ pairs as result.

MapReduce can be easily deployed and used. What users need to do is just implementing MAP and REDUCE functions. The implementation of MapReduce runs efficiently on a cluster of commodity machines and provides fault tolerance: a worker failure will not cause the whole job to be failed. It is also highly scalable that a MapReduce job typically processes terabytes of data.

talk about the advantage of map reduce: efficiency on commodity hardware, fault tolerance, scalable, etc.

3.1.2 Branch-and-Bound on MapReduce

If we take a look at the BnB search tree of the algorithm, we can observe that the expansions of nodes (each node is correspond with a state space) on the same level are independent. Therefore, the branch procedure can be done parallel for a level of nodes. After that the bound procedure is done on all nodes expanded.

This can be fit into the MapReduce model. In a MapReduce model, input data is a list of $(key, value)$ pairs, which is first processed by MAP function. The MAP function takes each $(key_1, value_1)$ pair as input, do some calculation on it and emits a list of $(key_2, value_2)$ pairs. The outputs are then grouped by keys, and sent to the REDUCE function. The REDUCE function takes a key and a list of values as input, and outputs a list of $(key_3, value_3)$ pairs.

In our design, we process each level of the search tree with one MapReduce job, where we call it one iteration. In the i-th iteration, we expand the i-th level of nodes. Therefore the whole search needs n iterations. For each iteration, the MAP function is designed as follows

Algorithm 2 Map

```
function MAP($Key, Value$)
    $S \leftarrow Value$
    $(S_1, S_2, ..., S_m) \leftarrow$ BRANCH($S$)
    for $i \leftarrow 1$ to $m$ do
        $GUB \leftarrow \min(GUB, \text{UPPERBOUND}(S_i))$
    end for
    $result \leftarrow \emptyset$
    for $i \leftarrow 1$ to $m$ do
        if LOWERBOUND($S_i$) $< GUB$ then
            $result \leftarrow result \cup S_i$
        end if
    end for
    return $result$
end function
```

Here Key is *Null*, and $Value$ is a state space node,

After each `map` function processes the brach(es) assigned, we need to allow all `map` functions to exchange their results to get global miminum upper bound. There are two major problems with this global upper bound update: 1) we need to wait for *all* upper bounds from the `map` functions. In other words, we need to apply a *barrier* after the `map` executions. The problem with barrier is that we the slowest tasks determine the overall speed of the entire system. 2) to compute the global minimum upper bound, we need to send all local bound to a single place, which results in a single `reduce` task, significantly lowering the level of parallelism.

In order to eliminate the two bottlenecks of parallel solution, we used the following two approaches. In the first approach we use what we call *random grouping* to solve this problem. We can retain full compatibility with existing MapReduce framework with this approach. In the second approach, we added a global state server to manage the global variable GUB and use asynchronous communication paradigm and eventual consistency to increase the level of parallelism. We discuss the first approach in Section 3.2 and second approach in Section 3.3.

3.2 Random Grouping Approach

Let a piece of input of map be $(key, value)$. We set key to be empty and let $value$ represent a node on the search tree, which includes $n+3$ fields. The first 3 states are GUB, LB and UB. The following n fields corresponds to the states of each vertex with -1 representing the vertex has not been searched yet.

In the map function we expand a node $k = \{GUB, LB, UB, x_1, x_2, \ldots, x_n\}$ to a list of nodes $L = \{k_1, k_2, \ldots\}$, calculate lower bound and upper bound for each k_i and finally update GUB for each extended node k_i. Thus the output nodes of each MAP function see a "local" view of GUB. We assign each output node's key field with a random integer which lies in $[0, r)$. The nodes with same key will be sent to the same reduce, and thus nodes will share their local GUB in order to get the global minimum of GUB.

THEOREM 1. *Let n be the total number of expanded nodes that their local GUB is the global minimum of GUB, and r be the number of groups that all nodes divided. Then all nodes will know the global minimum of GUB with probability at least $1 - n^{-c}$ if $r \leq \frac{n}{(c+1)\ln n}$ for any $c \geq 1$.*

PROOF. In order to make all nodes know GUB^*, which is the global minimum of GUB, each group need obtain at least one node of the above n nodes. Let A_i $(0 \leq i < r)$ be the event that there is no node knows GUB^* is allocated into the i-th group, then the probability of event A_i is

$$\Pr[A_i] = \left(1 - \frac{1}{r}\right)^n$$

and the probability $P[A]$ that at least one group does not get the above n nodes is

$$\begin{aligned} P[A] &= \Pr[A_0 \cup A_1 \cup ... \cup A_{r-1}] \\ &\leq \Pr[A_0] + \Pr[A_1] + ... + \Pr[A_{r-1}] \\ &= r\left(1 - \frac{1}{r}\right)^n \end{aligned}$$

By applying the inequality $1 - x \leq e^{-x}$ for all $x \in \mathbb{R}$, and the condition $r \leq \frac{n}{(c+1)\ln n}$, we get

$$\begin{aligned} P[A] &\leq re^{-n/r} \\ &\leq \frac{n}{(c+1)\ln n} e^{-(c+1)\ln n} \\ &= \frac{n^{-c}}{(c+1)\ln n} \\ &\leq n^{-c} \end{aligned}$$

The probability $\Pr[\bar{A}]$ that all nodes know GUB^* is

$$\Pr[\bar{A}] = 1 - \Pr[A] \geq 1 - n^{-c}$$

For instance, select $r = 200$ and $n = 10000$, then the probability is

$$\Pr[\bar{A}] \geq 1 - r\left(1 - \frac{1}{r}\right)^n = 1 - 200\left(1 - \frac{1}{200}\right)^{10000} \approx 1$$

3.3 Asychronous State Server Approach

With some experiments we notice that each iteration can be done with a map-only job, in which we omit the reduce part. This significantly reduces running time since the shuffle between map and reduce will sort the data, which we do not need. However, this will cause many nodes not seeing the global minimum upper bound, which leads to fewer nodes discarded than in random grouping. To solve this, we use a *parameter server*.

We set up a web server which stores the global minimum upper bound and supports query and modify to the bound. In each map, we start another thread, which constantly check if the local minimum upper bound is updated, and if so, this thread will interact with the web server. This thread is independent from the map function and thus will bring very little overhead. But with the server, the output nodes of each iteration reduces abount $1/3$ to $1/2$.

4 EXPERIMENTS AND RESULTS

In this section, we evaluate the performance and scalability of our system.

To demostrate the scalability of our algorithm, we show that we successfully solved four protein design problems that no other exact algorithms can do. Also we show that our system speed up linearly with the number of nodes. We also compare the performance of our distributed approaches with existing algorithms by repeating the same set of protein design problems. We can achieve significant speed up over single node solutions in a moderate-sized problems. Then we analyze the performance gains from various optimizations in our system using a set of micro-benchmarks. Finally, we demostrate the fault tolerance capability of our system.

All experiments, except for the single-node experiments, are performed on a cluster of XX nodes. Each node has two Intel Xeon E5-2620 (6-core, 2GHz) processors and 128GB RAM. Each nodes has three 3TB disks used in hadoop file system. The nodes are interconnected with 1Gbps ethernet. All nodes run commodity CentOS 6.5. Our system is implemented in Java on top of Apache Hadoop X.X. The state server is implemented using Ruby on Rails framework and runs on the same server configuration. The single node experiment is done on a node with 512GB RAM to allow the legacy single-node algorithm to scale as much as possible.

4.1 Scalability

4.1.1 Scale to a larger protein show the X problems that legacy algorithm cannot solve. analyze why (basically, how many branches are generated, and how much RAM it takes)

4.1.2 Speed-up with number of nodes

1-XX machines experiments, with all optimizations, showing linear speed up

4.2 Performance

In this section, we compare both approaches.

a table showing running time for different algorithms

explain the job setup overhead

analyzing the performance difference between random grouping and state server

We argue that although the state server approach is more effiecient than the random grouping approach, the latter has the advantage of being fully compatible with legacy MapReduce framework, and thus more applicable with Platform as a Service (PaaS) style services that only provide a MapReduce programming interface without allowing the `map` functions to contact the outside world.

4.3 Effectiveness of Algorithm Optimization

how to do this? how about disabling each optimization and see the results? (using a single design problem)

4.4 Fault Tolerance

kill nodes and show how much longer it takes.

kill the state server

5 CONCLUSION

ACKNOWLEDGEMENT

REFERENCES

Alvizo, O., Allen, B. D., and Mayo, S. L. (2007). Computational protein design promises to revolutionize protein engineering. *Biotechniques*, 42(1):31–35.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Dechter, R. and Rish, I. (2003). Mini-buckets: A general scheme for bounded inference. *Journal of the ACM (JACM)*, 50(2):107–153.

Gainza, P., Roberts, K. E., Georgiev, I., Lilien, R. H., Keedy, D. A., Chen, C.-Y., Reza, F., Anderson, A. C., Richardson, D. C., Richardson, J. S., et al. (2013). Osprey: protein design with ensembles, flexibility, and provable algorithms. *Methods in enzymology*, 523:87.

Globerson, A. and Jaakkola, T. S. (2008). Fixing max-product: Convergent message passing algorithms for map lp-relaxations. In *Advances in neural information processing systems*, pages 553–560.

Goldstein, R. F. (1994). Efficient rotamer elimination applied to protein side-chains and related spin glasses. *Biophysical Journal*, 66(5):1335–1340.

Hong, E.-J. and Lozano-Pérez, T. (2006). Protein side-chain placement through map estimation and problem-size reduction.

In *Algorithms in Bioinformatics*, pages 219–230. Springer.

Kingsford, C. L., Chazelle, B., and Singh, M. (2005). Solving and analyzing side-chain positioning problems using linear and integer programming. *Bioinformatics*, 21(7):1028–1039.

Kuhlman, B. and Baker, D. (2000). Native protein sequences are close to optimal for their structures. *Proceedings of the National Academy of Sciences*, 97(19):10383–10388.

Marinescu, R. and Dechter, R. (2009). And/or branch-and-bound search for combinatorial optimization in graphical models. *Artificial Intelligence*, 173(16):1457–1491.

Ottl, J., Battistuta, R., Pieper, M., Tschesche, H., Bode, W., Kühn, K., and Moroder, L. (1996). Design and synthesis of heterotrimeric collagen peptides with a built-in cystine-knot models for collagen catabolism by matrix-metalloproteases. *FEBS letters*, 398(1):31–36.

Pierce, N. A. and Winfree, E. (2002). Protein design is np-hard. *Protein Engineering*, 15(10):779–782.

Raha, K., Wollacott, A. M., Italia, M. J., and Desjarlais, J. R. (2000). Prediction of amino acid sequence from structure. *Protein Science*, 9(06):1106–1119.

Roberts, K. E., Cushing, P. R., Boisguerin, P., Madden, D. R., and Donald, B. R. (2012). Computational design of a pdz domain peptide inhibitor that rescues cftr activity. *PLoS computational biology*, 8(4):e1002477.

Rollon, E. and Dechter, R. (2010). Evaluating partition strategies for mini-bucket elimination. In *ISAIM*.

Rollon, E. and Larrosa, J. (2006). Mini-bucket elimination with bucket propagation. In *Principles and Practice of Constraint Programming-CP 2006*, pages 484–498. Springer.

Traoré, S., Allouche, D., André, I., de Givry, S., Katsirelos, G., Schiex, T., and Barbe, S. (2013). A new framework for computational protein design through cost function network optimization. *Bioinformatics*, 29(17):2129–2136.

Villalobos, A., Ness, J. E., Gustafsson, C., Minshull, J., and Govindarajan, S. (2006). Gene designer: a synthetic biology tool for constructing artificial dna segments. *BMC bioinformatics*, 7(1):285.

Voigt, C. A., Gordon, D. B., and Mayo, S. L. (2000). Trading accuracy for speed: A quantitative comparison of search algorithms in protein sequence design. *Journal of molecular biology*, 299(3):789–803.

Wainwright, M. J., Jaakkola, T. S., and Willsky, A. S. (2005). Map estimation via agreement on trees: message-passing and linear programming. *Information Theory, IEEE Transactions on*, 51(11):3697–3717.

Xu, J. and Berger, B. (2006). Fast and accurate algorithms for protein side-chain packing. *Journal of the ACM (JACM)*, 53(4):533–557.

Yanover, C., Meltzer, T., and Weiss, Y. (2006). Linear programming relaxations and belief propagation–an empirical study. *The Journal of Machine Learning Research*, 7:1887–1907.