A Distributed Computing Platform for Large-Scale Computational Protein Design

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 [**?**,]. People have successfully demonstrated the effectiveness of CPD in applications such as peptide synthesis [**?**,], protein-protein interactions [**?**,], artificial gene synthesis [**?**,].

(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 [**?**,]. On approach is to obtain approximate solutions, providing no guarantees on the optimal solution. This problem is modeled as a MAP-MRF inference problem [**?**,], which can be approximated by a Linear Programming Relaxation (LPR) problem [**?**,].

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 [**?**,], tree decomposition [**?**,], AND/OR Branch-and-Bound [**?**,], Integer Linear Programming [**?**,], Cost Network Function [**?**,].

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 MPLP [**?**,] and mini-bucket [**?**,] to compute the lower bound, and use Monte Carlo and simulated annealing to find a good 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 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 G-MEC 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 Overview

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.

2.1.1 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.

Algorithm 1 Branch-and-Bound Algorithm

```
1:  function BranchAndBound(G = (V, E), X, Θ)
2:      S_V ← X_1 × X_2 × ··· × X_n
3:      GUB ← UpperBound(S_V)
4:      Add(Q, S_V)
5:      while Q is not empty do
6:          S ← NextElement(Q)
7:          if LowerBound(S) ≥ GUB then
8:              continue
9:          end if
10:         (S_1, S_2, ..., S_m) ← Branch(S)
11:         for i ← 1 to m do
12:             GUB ← min(GUB, UpperBound(S_i))
13:         end for
14:         for i ← 1 to m do
15:             if LowerBound(S_i) < GUB then
16:                 Add(Q, S_i)
17:             end if
18:         end for
19:     end while
20:     return GUB
21: end function
```

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.

2.1.2 MapReduce

MapReduce is a programming model proposed by Google which is used for massive parallel and distributed computing on large data sets. It is composed of the following two main steps:

Map: The input data is divided into N parts. The code for processing each part and the data will be copied to a compute node.

Reduce: The result of *map* step is distributed to M compute nodes. The nodes run parallel and return the result.

What users need to do is just implement map and reduce functions. Map tells the platform how to process the data, and reduce tells how to combine the results.

2.1.3 Branch-and-Bound on MapReduce

If we take a look at the search tree of the algorithm, we can observe that the expansions of nodes 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, value)` pair as input, do some calculation on it and emits a list of `(key', value')` 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'', value'')` pairs.

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

When we come to the `reduce` function, we should iterate over all nodes expanded to get the minimum upper bound. This results in only one reduce in each job, which significantly reduces parallelism. We use what we call *random grouping* to solve this.

Algorithm 2 Map

```
1: function MAP(Key, Value, Context)
2:     S ← Value
3:     (S_1, S_2, ..., S_m) ← BRANCH(X)
4:     for i ← 1 to m do
5:         best ← min(GUB, UPPERBOUND(S_i))
6:     end for
7:     for i ← 1 to m do
8:         if LOWERBOUND(S_i) < GUB then
9:             Context.write(X)
10:        end if
11:    end for
12:    return best
13: end function
```

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 minimum upper bound, lower bound and upper bound. 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 $k = \{GUB, LB, UB,\ x_1, \ldots, x_n\}$ to a list of nodes $L = \{k_1, \ldots, k_m\}$, calculate lower bound and upper bound for each k_i and finally update the minimum upper bound for each k_i. Thus the output nodes of each map function see a 'local' view of the minimum upper bound. 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.

In the reduce function, nodes will share their local minimum upper bound. ????

Let t be the number of nodes expanded. Since we have r random integers, the nodes will be divided into r groups where each group has approximately $t_1 = \lceil t/r \rceil$ nodes. Assume there are q maps, normally the t nodes are equally produced by the maps. Therefore, at least $t_2 = \lceil t/q \rceil$ will see the actual global upper bound after map. The probability that none of the these nodes is in some particular group is

$$\left(\frac{r-1}{r}\right)^{t_2}$$

Let p be the probability that each node sees the actual global minimum upper bound after reduce. We

have

$$p \geq 1 - r\left(1 - \frac{1}{r}\right)^{t_2}$$

For instance, select r be 3-5 times the number of machines, probably 200. If we make t_2 be 10000, then we have

$$p \geq 1 - 200(1 - 1/200)^{10000} = 1$$

2.2 Optimization

2.2.1 Local Dead-End Elimination Algorithm

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 [**?**] 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

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

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*[**?**,]), we call the criterion in Eq. (3) integrated into the branch step the *local DEE criterion.*

2.2.2 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

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

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

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

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

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

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

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

The above optimization problem is a convex dual of MAPLPR, which can be solved by Convergent Message Passing Algorithms [**?**,].

If we compute the best functions β^*, then the lower bound of the current state space X becomes

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

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 $O(n^2 m^2)$, 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 $O(n^2 m^2)$ and space $O(n^2 m)$, then the time of computing LB_2 decrease to $O(n^2 m)$.

Mini-Bucket. The mini-bucket elimination (MBE) is a well-known approximation algorithm for graphical models [**?**, **?**, **?**,], 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.

2.2.3 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 [**?**, **?**,], and genetic algorithms [**?**,]. 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

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

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.

2.3 Refinement on MapReduce

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.

3 Results

4 Conclusion