BIOINFORMATICS

Vol. 00 no. 00 2015
Pages 1–5

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:

Contact: zengjy321@tsinghua.edu.cn

1 INTRODUCTION

In the past few years, Computational Protein Design (CPD) has become an important tool for protein engineering (Alvizo et al., 2007), such as peptide synthesis (Ottl et al., 1996), protein-protein interactions (Roberts et al., 2012), artificial gene synthesis (Villalobos et al., 2006), etc. In the structure-based computational protein design 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). 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 exists several methods which can solve the GMEC problem exactly, such as DEE/A* (OSPREY, Donald Lab at DUKE University), 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).

Our work is the first attempt to apply the branch-and-bound (BnB) search algorithm on distributed platform (such as MapReduce) to solve the Computational Protein Design problem. In our method, the DEE criteria 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 (Globerson and Jaakkola, 2008) and mini-bucket (Rollon and Dechter, 2010) to compute the lower bound, and use Monte Carlo and simulated annealing to find a good solution as our upper bound.

*To whom correspondence should be addressed.

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,

2.2 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 parts:

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 $GUB = \min_i UB(S_i)$, 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.

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(X_V)
  ADD(Q, X_V)
  while Q is not empty **do**
    $S \leftarrow$ NEXTELEMENT(Q)
    if LOWERBOUND(X) $\geq GUB$ **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 $best$
end function

Branch and Bound usually search the solution tree with BFS or optimum-cost-first method. Each live node has only one chance to be an expansion node, which produces all child nodes in one visit. The nodes which lead to infeasible or non-optimal solutions in the child nodes will be thrown away. The nodes left will be added to the list of live nodes.

2.3 MapReduce

MapReduce is a programming model proposed by Google which is used for massive parallel computing.

MapReduce is composed of the following two main steps:

1. Map procedure: The input data is divided into N parts. The code for processing each part and the data will be copied to a compute node. The compute nodes run parallel and return the result.
2. Reduce procedure: The result of step 1 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.

2.4 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

Algorithm 2 Map

function MAP($Key, Value, Context$)
  $S \leftarrow Value$
  $(S_1, S_2, ..., S_m) \leftarrow$ BRANCH(X)
  for $i \leftarrow 1$ **to** m **do**
    $best \leftarrow \min(GUB,$ UPPERBOUND(S_i))
  end for
  for $i \leftarrow 1$ **to** m **do**
    if LOWERBOUND(S_i) $< GUB$ **then**
      $Context.write(X)$
    end if
  end for
  return $best$
end function

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.

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 $k = \{GUB, LB, UB, state_1, \ldots, state_n\}$ 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

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

3 OPTIMIZATION

3.1 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 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) in 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

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

So that we can eliminate a large number of infeasible variable states, and thus significantly reduces the branch space.

Here, ...

3.2 Lower Bound

Naive Lower Bound. Firstly we have a naive lower bound, the heuristic function is

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

and the form of the current state space X is

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

there we leave $(u, v) \in E$ out from the summation notation for convenience, and $g(X)$ is the energy that has determined, that is

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

Efficient Lower Bound. By observing the formula of the naive lower bound, we see that every edge-energy function is only used for the vertex that has the greater index. 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

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

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

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

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

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 $O(n^2m^2)$. 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, and it gives a bound when the induced width of the graph is too large. The idea of bucket elimination (BE) is to eliminate variables, and the pseudo-code is as follows.

Algorithm 3 Bucket Elimination

```
function BucketElimination(F, v)
    B ← {f ∈ F | v ∈ var(f)}
    g ← B ⇓ v
    F ← (F \ B) ∪ {g}
    return F
end function
```

There the operation $\mathcal{B} \Downarrow v$ means combine the functions in $\mathcal{B}$ with eliminating the variable v. The time and space complexity of the algorithm is exponential in the largest scope of all the functions that computed.

Mini-bucket elimination has a parameter z which restrict the size of the scopes of each functions, and also restrict the time and space complexity.

Algorithm 4 Mini-Bucket Elimination

```
function BucketElimination(F, v)
    B ← {f ∈ F | v ∈ var(f)}
    (Q_1, Q_2, ..., Q_m) ← Partition(B, z)
    for i ← 1 to m do
        g_i ← Q_i ⇓ v
    end for
    F ← (F \ B) ∪ {g_1, g_2, ..., g_m}
    return F
end function
```

Partition$(\mathcal{B}, z)$ means that divide $\mathcal{B}$ into parts $Q_1, Q_2, ..., Q_m$ and $var(Q_i) \le z + 1$ for $1 \le i \le m$.

For the current state space X, we eliminate all the variables in $U(X)$, and then summarize all the functions on $D(X)$, then we get another lower bound of state space X.

3.3 Upper Bound

Different with lower bound, we make the upper bound of state space X be a better solution in X. There are many algorithm for finding a better solution of a problem, such as Simulated Annealing (SA), Genetic Algorithm.

We use Simulated Annealing as our upper bound algorithm. The initial state of SA is according to LB_2, which means

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

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

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

3.4 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 $1/2$.

4 RESULTS

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.

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.

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.

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*.

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.

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.