Nebula: A Blockchain Based Decentralized Sharing Computing Platform

Bin Yan, Pengfei Chen(✉), Xiaoyun Li, and Yongfeng Wang

School of Data Science and Computing, Sun Yat-sen University, Guangzhou, China
{yanb25,chenpf7,lixy223,wangyf226}@mail2.sysu.edu.cn

Abstract. Nowadays, there is a considerable amount of idle computers whose computing resources are partially wasted. On the other hand, the demand of resources is rapidly growing, since the explosion of data and the complexity of algorithms. To settle the contradictions, we develop Nebula, a decentralized platform based on blockchain for sharing computing resources. Nebula leverages blockchain to gather the scattered computing resources and provide a secure and vibrant computation trading market. Compared to traditional cloud platform, Nebula guarantees extra security because all transactions in this platform are validated by smart contracts. No one can tamper the transaction orders which are recorded by a widely distributed ledger. In Nebula, the resource consumer can order resources from resource providers with a very simple declarative script. When a deal is done, consumers can submit jobs to suppliers with a docker instance. Moreover, we model the order matching procedure of users' requests into a global maximum matching problem in a bipartite graph. We adopt the Hungarian algorithm to find an order matching policy, bringing an 10% increase to the matching rate in our best case. Moreover, we leverage the Proof of Authority (PoA) consensus algorithm called Clique, rather than Proof of Work (PoW) to increase the efficiency of Nebula, which provides nearly no less security but requires negligible computation on reaching consensus. To our best knowledge, we are the first to propose a general blockchain based platform for sharing computing resources, which fully utilizes the features of blockchain to achieve the scalability, the optimal order matching and a high performance.

Keywords: Blockchain · Cloud computing · Smart contract · Ethereum · Resource sharing

1 Introduction

Last decade has witnessed the growing demand of computational resources, as researchers have to deal with larger data and more complex algorithms. Especially with the advent of Big Data and Artificial Intelligence (AI), massive computation resources are needed including CPU and GPU. On the contrary, a considerable amount of devices, such as PC and servers in the data center are always running under low utilization [5,6,14]. For example, as stated by [5], 80% of servers in Google's data center are running under 20% utilization. These computers show a great potential to fill the requirement gap but fail to gather together and being scheduled. Highly distributed devices around the world currently are not reliably connected and arranged to meet the requirement of the users.

As for organizing the public computing resource, some systems such as BOINC [1], SETI [18] are successful pioneers. However, the computing resources are joined in a volunteer way which makes them are unstable. Moreover, they are designed for some specific jobs especially for science computing. People need a long-time learning curve to run their jobs in such a platform. Recently, the blockchain technology has been proposed and widely studied in both of academic world and industry world [2–4,12,15,16,21,22]. From one survey [23], we can see there are many fields where Blockchain can be applied.

Due to the tamper-resistant, security, and token based ecology of Blockchain, it has been widely used in many fields such as food tracking, currency exchange and so on. Recently, a novel kind of cloud computing named decentralized cloud computing driven by Blockchain emerges such as Golem [10], iEXEC [11], SONM [19], UChain [20]. Although they are capable to run decentralized applications on their chains, they are not sufficient to support general computing. SONM [19] is the most similar to our paper. However, it does not provide an global optimal order matching policy, which leads to a low efficiency. To integrate the idle resources and overcome the drawbacks of existing systems, we develop Nebula, a decentralized platform based on Ethereum [7] for computational resources sharing.

Two main problems should be resolved for any sharing platform like Nebula. The first one is security. In our implementation, smart contract, as a validator, ensures every attempt to modify the system (e.g. a user's request) is legal and all the data accepted by the system is consistent. The blockchain technique itself further makes sure that data is unmodified and trusty.

The second one is stimulation. Nebula, like an ecosystem of *Sharing Economy*, should fairly reward suppliers and charge consumers. Nebula leverages a cryptocurrency called Nebula Token as a payment of computational resources. Nebula Token is free to transfer to/from Ether (i.e., Ethereum Token) [7] and for every transaction an indicative price will be proposed, which both reduce the fluctuation of its value.

Nebula has a specially designed architecture. It fully utilizes the feature of blockchain technique. All the business logic functions of the system are written in smart contracts [13], which are mainly deployed in our customized blockchain, the *sidechain*. The sidechain connects with Ethereum, or the *mainchain*, by a smart contract called *channel*. Other peripheral components including Data Cache (DC), NAT-service (Network Address Transformation service), etc, provide extra features or support to the system, as described in detail in Sect. 3.

In the matching step between suppliers and customers' orders, we model orders into a bipartite graph and apply Hungarian algorithm to promote the matching rate. After the optimization, the matching procedure costs nearly the same time but gets an increase of 10% matching rate in our case.

The contributions of this paper are summarized as follows. **First**, we introduce Nebula, the blockchain based decentralized platform for sharing computing resources. We present the components, their functions and the utilization of the blockchain technique in detail. **Second**, we show the optimization of the order matching procedure and the performance gain by experiments.

The rest of the paper is organized as follows. In Sect. 2 we discuss the motivation and show the overview. In Sects. 3 and 4, we talk about the detailed implementation of the system. Then we evaluate our system in Sect. 5, including the cost of matching procedure and the throughput of the system. At last, we conclude the paper in Sect. 6.

2 Motivation and System Overview

Our motivation is around the contradictions between current requirements of computing and the idle computing resources. In recent years, data has been growing explosively fast and the algorithm has become much more complex. They bring great challenges to researcher's available computing resources. It is not practical for individuals or organizations to purchase machines infinitely to come up with the demand.

On the other hand, a considerable amount of devices around the world are idle, indicating great potential to fill the requirement gap. The solution to the contradiction is a platform which gathers and schedules idle devices, regardless of their locations, types and performance. Next, we introduce Nebula in a nutshell, a decentralized platform based on Ethereum [7] for sharing computing resources.

2.1 Decentralization

Decentralization is an important feature of Blockchain. It allows trading between peers without centralized organizations. The decentralization presents in two ways. (i) **The backend**. Nebula sits on Ethereum, a distributed peer-to-peer platform. It provides a high availability and reliability and maintains the validity of data. (ii) **The resources**. Nebula is more of an agency than a provider because the resource Nebula provided is owned by end users from all over the world. Some of users, called suppliers, contribute their devices into the system with Nebula Token as a reward, while other users, called customers, pay for the usage of devices with Nebula Token.

2.2 Ethereum Based System

As a Ethereum-backed system, Nebula fully utilizes the feature of Ethereum, such as the smart contract technique, the digital currency and its immutability of data.

Smart Contract. The smart contract is a major component of Ethereum, intended to digitally enforce the performance of codes. It is deployed to the blockchain by the administrator after the initialization of the blockchain. The deployed contract is stored as a transaction in the system, hence it is immutable and transparent. In Nebula, the smart contract implements almost all the business logic. *The Market contract* especially accepts users' orders and validates requests. Other contracts support, for example, the persistent storage of orders, the validation of deals and a channel for token circulation as described in Sect. 3, etc.

Digital Currency. Ether is a fundamental token for the operation of Ethereum. Based on Ether, Nebula introduces *Nebula Token* as a payment for transactions between buyers (i.e., customers) and sellers (i.e.,suppliers). Nebula proposes an indicative price for every transaction and fix the exchange rate between Ether and Nebula Token to avoid fluctuation. The indicative price is given by a regression model based on our history transaction prices. We leave it as a future work to build a more sophisticated and accurate model. Nebula Token will be the main stimulation for device owners to join the system.

Immutability. All transactions on the Ethereum blockchain are immutable. Any manipulated transaction invalidates the PoW (Proof of Work) and thus will be rejected. Nebula utilizes this kind of immutability and stores all the transaction data in the blockchain.

3 Detailed Design of Nebula

In this section, we demonstrate the design of the system in detail. We show the architecture of the system and introduce important smart contracts, especially the *market* and *channel* contracts. Then, we describe the workflow of token circulation. Lastly, we show the optimization of the order matching procedure and then describe and analyze the Hungarian algorithm.

```
time: 2h               ----------------------------  the order duration
price: 0.1 RMB/h       ----------------------------  the price of this order
resources:             ----------------------------  define the buy/sell resources
  network:             ----------------------------  network resource
      bandwidth: 2 Mbits/s
  cpu:                 ----------------------------  the number of cpu cores
      cores: 1
  memory:              ----------------------------  the volume of memory
      size: 1GB
  gpu:                 ----------------------------  the number of GPU
      num: 1
```

Fig. 1. An example of declarative script for resource demand.

3.1 Architecture

We describe how the user interacts with the system before introducing the step to launch the system and the function of peripheral services.

Usage. Here we introduce the typical workflow of users. Firstly, (a) the customer should describe his demand of resources with a declarative script. The script expresses as a YAML file shown in Fig. 1. After scripts being parsed by the *client*, the order is placed to the system with the help of the *node service*, which hides the complex interaction. Then, (b) the supplier can join his machine by running the *worker service*, which benchmarks the device and establishes a connection to Nebula. The supplier is required to explicitly confirm his devices, and sends his request of provision the same way as the customer does. Finally, if the supplier's provision is satisfactory to the customer's demand, a deal is done and each side will be notified. Then the customer can submit tasks to the supplier's worker machine with a docker instance.

Launch of Nebula. Launching Nebula includes building the blockchains, deploying the contracts and starting the significant services.

Nebula is built on top of two blockchains, a *mainchain* and a *sidechain.* The mainchain essentially is the Ethereum. The sidechain is a Nebula-specific blockchain where our contracts are deployed. A separated sidechain isolates a stable and controllable environment to operate and maintain the system. To allow token to circulate between blockchains, a channel contract is deployed onto both blockchains as a connector. Other indispensable *peripheral services* serve differently as described below in detail.

Figure 2 shows the main architecture of Nebula. To launch Nebula, (1) we firstly build the sidechain and deploy the smart contracts in the sidechain. Particularly, the channel contract is also deployed and the circulation between the mainchain and the sidechain is thus established. Then, (2) the *DC (Data Cache)* service and (3) the *NAT-service* are launched in sequence and the system is now ready to response to users' request.

Peripheral Services. Here we introduce other building bricks of Nebula, DC and the NAT-service. Every peer in the sidechain is required to maintain a cache service called DC (Data Cache), in order to reduce the response delay. DC is essentially a server with a database, which caches and tracks the state of the sidechain, responses the query from users and asynchronously sends requests to the sidechain on behalf of the users.

Nebula also contains NAT-service for users across different subnets to connect with each other. The service also allows the supplier to control their worker machines in a different subnet.

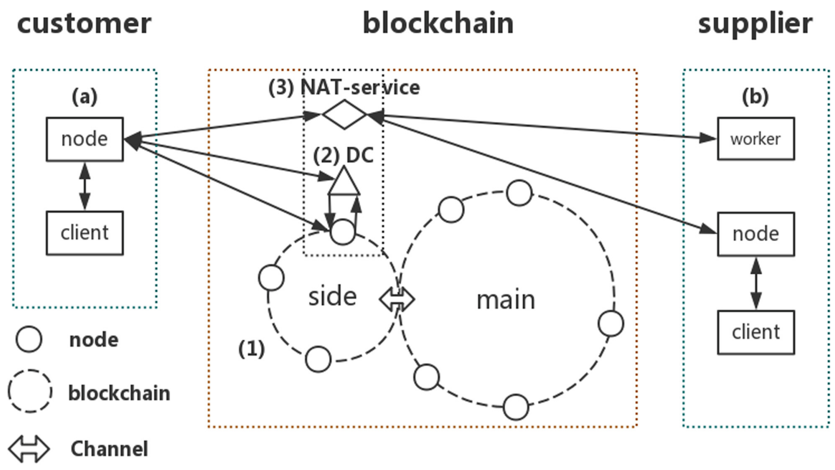

Fig. 2. Nebula architecture. Mainchain stands for Ethereum. Sidechain is a Nebula-specific blockchain where contracts are deployed. Two blockchains are interacted by a channel.

3.2 Networking

In practice, the customer and the supplier may not locate in the same subnet. Even worse, the machine owner and his cluster are likely to be separated in the network, which prevents the owner from managing his devices. To support the cross-network communication, Nebula provides an *NAT-service* to establish a tunnel, by which the real-time communication can be performed.

3.3 Consensus Algorithm

We leverage PoA rather than the traditional PoW algorithm to achieve consensus in the sidechain, achieving a much less mining delay and reducing the waste of resources. We choose the *Clique PoA* algorithm as it is used stably for years in the famous Rinkeby testnet [17].

In Clique, a block is authored if it is signed by a peer from the list of authorized signers. The cost of signing is negligible compared to the PoW computation. Every signer is only allowed to sign one out of *SIGNER-LIMIT* consecutive blocks to protect the network from being damaged by the malicious user. The authorized user list is set in the genesis block, and changes as users being voted in or out.

3.4 Smart Contract

Computation Market. The smart contract *Market* implements the market of the system, where orders are accepted, validated, stored and finally matched one-to-one. Various functions are defined in the Market contract that handle the creation, cancelation and modification of the orders, as well as the joining and exiting of devices.

Listing 1.1. Pseudocode Code of function PlaceOrder

```
    contract Market {
 2      ...
        function PlaceOrder(
 4          UserType userType,
            uint duration,
 6          uint price,
            uint[] benchmarks
 8      ) returns (uint) {
            /* omit validation codes here */
10          if (userType == UserType.CUSTOMER) {
                uint lockedSum = calculate payment for an hour;
12              if (fail to transfer lockedSum token to msg.sender) {
                    error("failed to prepay a unit of token");
14              }
            }
16
            ordersAmount = ordersAmount + 1;
18          uint orderId = ordersAmount;
            orders[orderId] = Order(
20              userType, msg.sender,
                duration, price, benchmarks,
22          );
            emit OrderPlaced(orderId);
24          return orderId;
        }
26  }
```

Listing 1.1 shows the function *PlaceOrder* as an example, the handler for the creation of orders. The function accepts parameters like a user type (one of supplier or customer), the duration of the task, the bid price and a set of benchmarks specifying the amount of resources. After the place-order request is sent to the system, the validation will be performed at the earliest (omitted in the listing codes). Then a unit of token is prepaid to the intermediate account as an advance fee (Line 10–14). Otherwise the request will be rejected if the transfer fails. Next, the order is persistently stored into the system (Line 18–24). Lastly, the function call succeeds, as an event is emitted and the index of the order (*orderId*) is returned.

Channel and Nebula Token. A special method named channel is applied to allow token circulation between mainchain (Ethereum) and our sidechain. A token circulation from Ethereum to the sidechain takes the following steps, as illustrated by Fig. 3.

– The user transfers his Ether to the channel account on Ethereum.
– The Nebula administrator(a privileged daemon) notifies the channel on the sidechain.

– The sidechain channel, with a nearly infinite amount of Nebula Token obtained on deployment, finally transfers the equivalent Nebula Token to the user.

The user is free to get his Ether back to Ethereum in the opposite way, or exchange between Nebula Token and Ether on the sidechain.

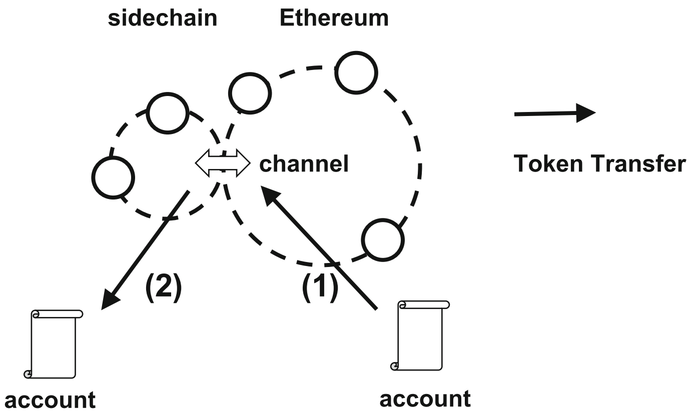

Fig. 3. Demonstration of token circulation between Ethereum and our sidechain.

3.5 Order Description

Users can buy or sell the resources by sending an *order* to the system. An order consists of a plenty of fields including the user type, the duration, the bid price and the specification of the resources, such as the number of CPU cores, the capacity of the memory and the storage device, etc. We generally embed the order into a vector in the order assigning procedure (see Sect. 3.6), with the user type discarded.

3.6 Order Assigning

Order assigning is the procedure to assign a satisfactory supplier's order to a customer's one. We say a supplier's order is *satisfactory*, if the duration and the resources offered by the supplier is no less than what the customer requests. Moreover, the price from the supplier is lower than that from the customer.

The orders arrive in the system continuously. To simplify the design of the matching step, orders are buffered before issued to the matching algorithm batch by batch. The order which fails to find his counterpart by the algorithm will be recalled to the buffer and wait for the next try.

To obtain the maximum matching rate, we transform the batch of orders into a bipartite graph and adopt the Hungarian algorithm to find the perfect (maximum) matching.

Order Embedding and Graph Construction. A customer's or supplier's order is turned into a vector $V_C \in \mathbb{R}^n (V_S \in \mathbb{R}^n)$, with the duration, the bid price and the resources included.

Our ultimate goal is to match orders between two groups of users, namely customers and suppliers. It is equivalent to find a maximum matching of a bipartite graph between two vertex sets, the customer's vector set and the supplier's vector set.

Here we show Algorithm 1 to construct the bipartite graph. Initiate the graph with empty edge and vertex (line 1). For each vector provided as input, add a corresponding vertex to the graph representing the vector (line 2–7). For each vertex pair (V_C^i, V_S^j) from customer's vertex set and supplier's vertex set, we examine whether V_S^j is satisfactory to V_C^i, i.e. $V_C^i \prec V_S^j$. The *partial order* is defined over the vector as Definition 1. The returned result by Algorithm 1 is a bipartite graph.

Definition 1. *For any two vectors V, $U \in \mathbb{R}^n$, we define $V \prec U \iff \forall i \in \mathbb{Z}$, $V_i < U_i$, $0 \leq i < n$*

Algorithm 1. Graph Construction

```
Data: customer's order embedding matrix $M_C = [V_C^1 \cdots V_C^n]$ and supplier's one
      $M_S = [V_S^1 \cdots V_S^m]$
Result: a constructed bipartite graph
$G \leftarrow (V, E)$ ;      // G is an empty Graph denoted by empty vertex $V$ and
edge $E$
for $V_C^i$ in $M_C$ do
    $V.add(V_C^i)$;
end
for $V_S^i$ in $M_S$ do
    $V.add(V_S^i)$;
end
for $V_C^i$ in $M_C$ do
    for $V_S^i$ in $M_S$ do
        if $V_C^i \prec V_S^j$ then
            $E.add((V_C^i, V_S^j))$;
        end
    end
end
return G
```

Hungarian Algorithm. Finally, we adopt Hungarian algorithm (Algorithm 2) to find the maximum matching. Figure 4 shows a tiny example for the algorithm. The idea behind it is simple. For every vertex in a part, we try to find its

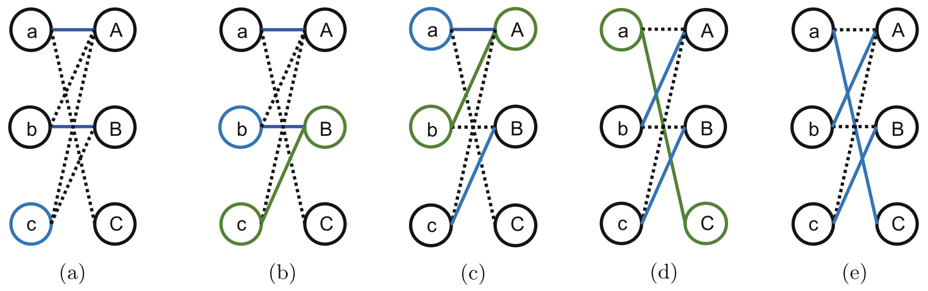

Fig. 4. Hungarian algorithm example. (a) Initially “a” matches “A”, “b” matches “B”. “c” is left over and to be matched. (b) There is no available counterpart for “c”, so “c” preempts “b” and gets its counterpart “B”. (c) There is no available counterpart for “b”, so “b” preempts “a” and gets its counterpart “A”. (d) “a” has another available counterpart “C”, so “C” is assigned to “a”. (e) The final result of the matching.

counterpart in the other part. If there are available counterparts, assign one of them to the vertex. If all of its candidate counterparts have been assigned, we “grab” one of the unavailable counterpart and recursively try to find a new counterpart for the unlucky grabbed one. Figure 4(b)–(d) illustrates the process in detail.

Algorithm 2. Hungarian algorithm

```
Data: adjacency matrix of a bipartite graph $G \in \{\text{True}, \text{False}\}^{n\times n}$
Result: a counterpart array $C \in \mathbb{Z}^n$
1 $C \leftarrow [-1 \cdots -1]$;                         // -1 means counterpart not found
2 $V \leftarrow [\text{False} \cdots \text{False}]$;  // V[i] represents whether the i-th node has been
  visited
3 $i \leftarrow 0$;
4 while $i < n$ do
5     $V \leftarrow [\text{False} \cdots \text{False}]$;
6     $Find(i)$;
7     $i \leftarrow i + 1$;
8 end
9 return C
```

Here we calculate the time complexity of the matching step. The graph construction step compares every pair of orders from two sets, thus it costs $O((\frac{n}{2})^2) = O(n^2)$, where n is the total number of orders. It is easy to show that the complexity of the Hungarian algorithm is $O(V \cdot E)$, noting that for every vertex in the graph, *Find* costs no more than $O(E)$ and *Find* is invoked for every vertex. Thus the overall complexity of the matching is $O(V^2 + VE)$, where $O(n) = O(V)$.

The complexity of the matching step is almost the same as the naive First Fit algorithm, especially when the number of order is under a couple of thousands. However, the Hungarian algorithm raises the matching rate by 10% in the best case, as shown in Sect. 5.

Deal Opening. When two orders are matched, a corresponding *deal* is opened to record the matching relationship of the orders. Listing 1.2 shows the implementation of function *OpenDeal*, which handles the validating and recording of the deal. The function firstly checks the consistency of the opening deal, such as the user types, the constraint of price, duration and the benchmarks (Line 7–13). The constraint of the numeric fields should be trivially satisfied, since the order assigning procedure only matches satisfactory ones. After passing all the requirement check, the deal is recorded on the blockchain and an event is emitted to mark the success.

3.7 Delivery of Computing Resources

The computing resources are delivered by means of running the customer's task on the supplier's working machine. To get his working machine ready, the supplier (1) binds the address of the working machine with his personal account, and (2) places a selling request to the system. After the orders are matched between the customer and supplier, both users are able to query their counterpart and the status of the order from the command line.

To assign a task to the working machine, the customer (1) builds a docker image which contains the code and its corresponding data, and (2) uploads the image to a public image repository like Docker Hub. Then, the customer (3)

Algorithm 3. Find

```
Data: index $i \in \mathbb{Z}^+$ of a node to look for its counterpart
Result: True if succeed, False otherwise
1  for $t \leftarrow 0;\ t < n;\ t \leftarrow t + 1$ do
2      if $G[i][t]$ and $!V[t]$ then
3          $V[t] \leftarrow$ True;
4          if $C[t] == -1$ or $Find(C[t])$ then
5              $C[i] = t$;
6              $C[t] = i$;
7              return True
8          end
9      end
10 end
11 return False
```

informs the working machine to run the task by providing the image repository and the image tag. Notified by the request, a piece of script in the working machine pulls and runs the image and sends the execution log back to the customer.

The following reasons make us choose docker to deliver tasks. **First**, docker provides stable operation. A docker image packs up the code with the supportive environment, which conceals the uncertainty of the counterpart machine. **Second**, docker has negligible run-time overhead compared to the virtual machine. The containers share the machine's OS system kernel and therefore do not require an OS per application. **Third**, the cgroups technique allows controllable delivery of resources. cgroups is a Linux kernel feature that limits and isolates the resource usage of a collection of processes. Every task launched by Nebula is limited to what written in the supplier's provision order.

3.8 Payment Strategy

The payment strategy mainly focuses on credibility and fairness. Credibility is obviously realized, since Nebula is built upon Ethereum and mainly implements its business logic in smart contracts. We will talk about achieving fairness in detail. Here we define fairness as nobody is able to earn an extra benefits (token or computing resources) by any means.

When a customer places an order to the system, an advance charge will be paid to an intermediary address. If the prepay fails, the transaction will be reverted by the sidechain and thus the request is rejected. After the task begins, tokens will be continuously transferred from the customer to the supplier with the intermediary address as an agent. Only after the working machine finishes his round, the token will be transferred to the supplier's account. If the order is illegally canceled, the prepaid token will be sent to his counterpart as a compensation. Because of this prepaying strategy, a dishonest user is not able to steal tokens/resources by any means.

4 Implementation of Nebula

Nebula is a complicated system. The implementation of Nebula involves various programming languages like Go and Python and open source tools. The components are interacting by gRPC, a language-neutral framework with high performance to perform remote procedure calls. The number of code has exceeded 50 thousands of lines.

Listing 1.2. Pseudocode Code of function OpenDeal

```
contract Market {
    ...
    function OpenDeal(uint buyID, uint sellID) {
        Order buy = orders[buyID];
        Order sell = orders[sellID];

        require(buy.userType == UserType.COSTUMER);
        require(sell.userType == UserType.SUPPLIER);
        require(sell.price <= buy.price);
        require(sell.duration >= buy.duration);
        for (i = 0; i < sup.benchmarks.length; i++) {
            require(sell.benchmarks[i] >= buy.benchmarks[i]);
        }

        dealAmount = dealAmount + 1;
        deals[dealAmount] = Deal(
            sell.benchmarks, _sellID, _buyID, buy.duration, sell.price
        );
        emit DealOpened(dealAmount);
    }
}
```

4.1 Services

Most components in Nebula are designed as services to make it accessible by the third-part software. For example, the node service, the DC service and the worker service are shown in Sect. 3. All services in Nebula are implemented in Golang. The reason we prefer it to other languages like C++, Java is that it is efficient and it naturally supports multi-threading, which is convenient for asynchronous communication. Golang proves to be a good language to implement a service in practice.

4.2 Smart Contract

The smart contracts are written in solidity [7] with version ˆ0.4.20. In the production environment, we use Geth [9] to build a blockchain and use Truffle to compile and deploy the contract.

We choose a mature solution called Truffle Suite, a group of open-source tools for the blockchain developer. Included in Truffle Suite, Ganache [8] is used to build the mock blockchain, Truffle is used to compile, deploy and debug the contracts, and Dizzle is used to develop the frontend of the web interface.

4.3 Web Interfaces

Nebula provides a GUI interface based on web to help non-programmers to interact with Nebula. The backend for the website is essentially some of the services or components mentioned in Sect. 3, such as DC and sidechain.

5 Experimental Validations

5.1 Throughput of Sending Orders

The major delay of sending an order comes from mining in the blockchain. Besides, other factors affect the delay, including the state of the network, the performance of the machine and so on. To avoid the influence and reveal the standard performance of Nebula, we instead show the relationship between the block interval and the average delay of placing an order.

We build the blockchain using Ganache with the parameter *blockTime* set ascendingly from 0 s to 16 s, which simulates the various mining delay in practice. As a comparison, the mining delay is 10 s–20 s in the Rinkeby testnet.

We sequentially place orders and record the average response time. To test the throughput, we concurrently send orders to the system and record the average delay. Figure 5 shows the experiment result. It is obvious that the response time approximates to the block interval, since a request is acknowledged only after the corresponding block has been signed. The throughput is much higher than the serial case, with an increase of 43% in the best case.

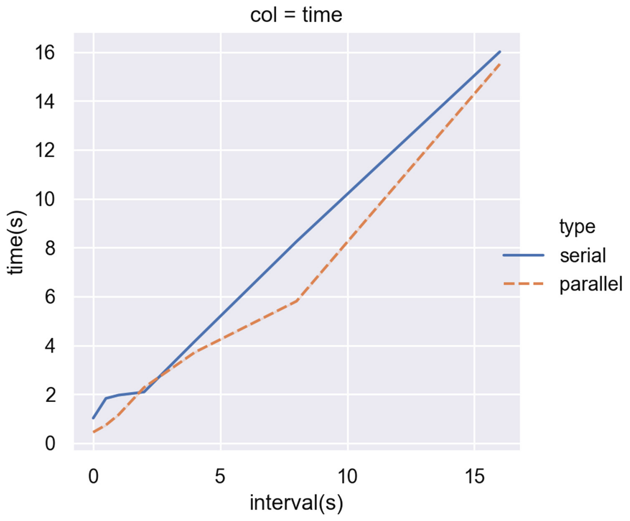

Fig. 5. Throughput of Nebula.

5.2 Matching Rate of Hungarian Algorithm

We design experiments to compare the successful matching rate of Hungarian algorithm against two benchmarks, the First Fit algorithm and Best Fit algorithm.

We firstly show the implementation of the benchmarks. The First Fit algorithm assigns the *first* satisfactory supplier's order to the customer's one. The Best Fit algorithm assigns the *closest* order among the satisfactory ones to the customer. The distance between orders is measured by the euclidean distance between two corresponding embedded vectors.

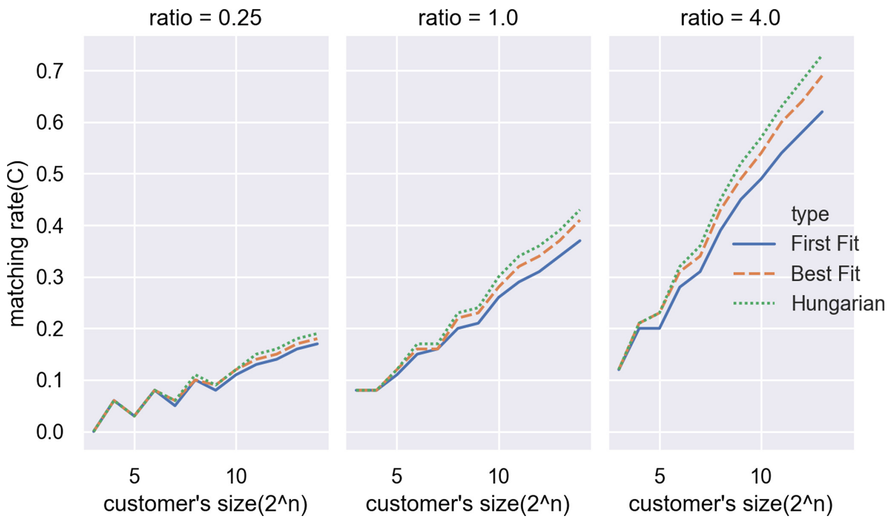

Fig. 6. Matching rate of the customers' order. ratio = number of suppliers/number of customers

Figure 6 shows the matching rate of customers' order under the variety of the number of supplier's order. The matching rate of the Hungarian algorithm is always the highest among any other algorithms, as it is capable to find the maximum matching.

When the number of customer's orders is about four times than the supplier's, the matching rate (for the supplier's order) among the algorithms is nearly the same, since its counterpart's number is adequate and the naive algorithm could perform just as well. However, when the number of supplier's orders exceeds the customer's, the Hungarian algorithm shows huge performance improvement compared to the naive algorithms. Inspired by the result, Nebula will automatically choose between the Hungarian algorithm and Best Fit algorithm, according to the ratio of the orders. A higher matching rate avoids the rematch of the orders and thus can highly speed up the matching step.

6 Conclusion

This paper introduces Nebula, a blockchain based decentralized platform for sharing computing resources. To our best knowledge, Nebula is the first platform that fully utilizes the features of blockchain and avoids the weakness by peripheral services, such as data cache and the NAT-service. We emphasize on the novel architecture with two blockchains, i.e., the Ethereum and the sidechain. We also show how other peripheral services are deployed upon the sidechain. Lastly, we examine the throughput of placing orders and shows the average delay. We also present the adoption of the famous Hungarian algorithm and the 10% improvement of matching rate in our best case. The system is still in an initial stage. The design of the system will be improved in the future work. For example, we can apply an enhancement of the Hungarian algorithm, the Kuhn-Munkres algorithm, to the order matching procedure.

Acknowledgments. The work described in this paper was supported by the National Natural Science Foundation of China (61802448, U1811462) and the Program for National Natural Science Foundation of Guangdong (2019A1515012229). The corresponding author is Pengfei Chen.

References

1. Anderson, D.: BOINC: a system for public-resource computing and storage, pp. 4–10, December 2004. https://doi.org/10.1109/GRID.2004.14
2. Chen, W., Wu, J., Zheng, Z., Chen, C., Zhou, Y.: Market manipulation of bitcoin: evidence from mining the Mt. Gox transaction network. In: IEEE INFOCOM 2019-IEEE Conference on Computer Communications, pp. 964–972. IEEE (2019)
3. Chen, W., Zheng, Z., Cui, J., Ngai, E., Zheng, P., Zhou, Y.: Detecting Ponzi Schemes on Ethereum: towards healthier blockchain technology. In: Proceedings of the 2018 World Wide Web Conference, pp. 1409–1418 (2018)
4. Dai, H., Zheng, Z., Zhang, Y.: Blockchain for Internet of Things: a survey. CoRR abs/1906.00245 (2019). http://arxiv.org/abs/1906.00245
5. Delimitrou, C., Kozyrakis, C.: Quasar: resource-efficient and QoS-aware cluster management. In: 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2014)
6. Delimitrou, C., Kozyrakis, C.: Paragon: QoS-aware scheduling for heterogeneous datacenters, vol. 41, pp. 77–88, May 2013. https://doi.org/10.1145/2490301.2451125
7. Ethereum. https://www.ethereum.org/
8. Ganache. https://www.trufflesuite.com/ganache
9. Geth website. https://github.com/ethereum/go-ethereum/wiki/Geth
10. Golem. https://golem.network/
11. Iexec. https://iex.ec/
12. Li, Z., Kang, J., Yu, R., Ye, D., Deng, Q., Zhang, Y.: Consortium blockchain for secure energy trading in industrial Internet of Things. IEEE Trans. Industr. Inf. **14**(8), 3690–3700 (2017)
13. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, pp. 254–269. ACM, New York (2016). https://doi.org/10.1145/2976749.2978309
14. Mars, J., Tang, L., Skadron, K., Soffa, M.L., Hundt, R.: Increasing utilization in modern warehouse-scale computers using bubble-up. IEEE Micro **32**(3), 88–99 (2012). https://doi.org/10.1109/MM.2012.22

15. Nakamoto, S., et al.: Bitcoin: a peer-to-peer electronic cash system (2008)
16. Qiu, X., Liu, L., Chen, W., Hong, Z., Zheng, Z.: Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing. IEEE Trans. Veh. Technol. **68**(8), 8050–8062 (2019)
17. Rinkeby testnet. https://www.rinkeby.io/#stats
18. Seti@home. https://setiathome.ssl.berkeley.edu/
19. Sonm. https://sonm.com/
20. Uchain. https://uchain.world/
21. Wang, J., Wang, H.: Monoxide: scale out blockchains with asynchronous consensus zones. In: 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19), pp. 95–112 (2019)
22. Zheng, P., Zheng, Z., Luo, X., Chen, X., Liu, X.: A detailed and real-time performance monitoring framework for blockchain systems. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 134–143. IEEE (2018)
23. Zheng, Z., Xie, S., Dai, H.N., Chen, X., Wang, H.: Blockchain challenges and opportunities: a survey. Int. J. Web Grid Serv. **14**(4), 352–375 (2018)