Distributed Parallel VN Embedding Based on Genetic Algorithm

Qiao Lu, ChangCheng Huang
Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada
Email: qiaolu@cmail.carleton.ca, huang@sce.carleton.ca

***Abstract*—Network virtualization has emerged to replace traditional network architecture since it allows multiple virtual networks to share a common substrate network. However, one of the main challenges for network virtualization is the resource allocation for each virtual network (VN), called Virtual Network Embedding Problem. The computation complexity of existing resource allocation approaches is too high to achieve an optimum within an acceptable time. Further, the provided optimum is not optimal in an online non-reconfigurable VN embedding setting because of the highly dynamic nature of user demands. Nowadays, due to lower hardware costs, distributed parallel computing can be used to deal with complex computing tasks with high efficiency. In this paper, we propose a distributed parallel Genetic Algorithm (GA) for solving VN Embedding problems. Through theoretical analysis, we compare the time saving of our distributed parallel algorithm with traditional sequential running. Results show that our algorithm achieves better performances on execution time and acceptance ratio.**

I. Introduction

Allocating physical resources in network virtualization to achieve higher profit and efficiency is a well-known problem. Such a problem is called Virtual Network (VN) Embedding. To solve the problem, algorithms need to decide the physical resources for VNs, which becomes an optimization problem with a particular objective (*e.g.,* lower cost). In general, a virtual network is composed of several virtual nodes and their associated virtual links. A common approach is to adopt two phases to allocate a virtual network: 1) node mapping: mapping virtual nodes into substrate nodes and 2) link mapping: mapping virtual links into substrate paths with multiple connected substrate links.

In an online VN Embedding problem, VN requests arrive dynamically and stay in the network for random durations. In most real-life scenarios, the VN Embedding is an online problem, which requires a speedy and efficient solution. Both offline and online VN Embedding can be formalized into an exact optimization problem, such as Integer Linear Programming (ILP) model [1]. However, exact solutions are time consuming and hard to achieve real global optimization under dynamic demands. Specifically, to avoid causing disruptions to the existing services, these solutions do not try to reallocate mapped requests in the past. Nor do they try to predict future requests. When a VN request arrives, they allocate the request based on current residual resources. Obviously, this kind of local optimal results may not lead to global optimization of resources usage. This observation provides space for heuristic algorithms to play.

Existing literature proposes several heuristic algorithms in VN Embedding to increase placement efficiency as well as improve global optimality. Generally, heuristic algorithms[1][2] tend to find one solution that is likely to be good. However, the performance of the solution is not guaranteed.

Few researchers propose meta-heuristic algorithms for VN Embedding problems. To our best knowledge, the previous papers[3][4] on Genetic Algorithm in VN embedding are only focused on node mapping. However, the more complicated VN link mapping is not well investigated yet.

In some research papers[1][5], it was assumed that a virtual node/link can be split into different substrate nodes/links, which is called splittable mapping. The embedding of a virtual network for both splittable and unsplittable cases is a very important research topic for software defined network (SDN) and network function virtualization (NFV). Indeed, the splittable embedding attains better resource utilization in theoretical analysis. Actually, the splittable embedding is easier than an unsplittable embedding[6]. The splittable embedding problem can be formulated as a linear program which can be solvable in polynomial time. However, the splittable embedding is hard to implement into a real online embedding problem.

Firstly, the splittable embedding creates the overhead of maintaining the state consistency, which requires more consideration for future work[6]. Secondly, mapping a virtual link to multiple paths in the substrate network may cause out-of-order packet delivery[2]. Out-of-order delivery is a primary concern for packets in the same flow. Some strategies such as Equal-cost Multi-path (ECMP) routing should be brought into implementation to deal with packet reordering as well as balancing multiple flows over multiple paths. Additionally, for the online embedding problem, we consider the time consuming as a very important factor. Polynomial time complexity is still far longer than we expect in an online embedding system.

The unsplittable VN embedding problem is a fundamental mathematical problem for provisioning resources for network slices[6]. Unfortunately the complexity problem becomes one major bottleneck of the unsplittable embedding. In previous work, unsplittable link mapping algorithms that are formulated as Integer Programming, are inclined to simplify the NP-hard problem to shortest path problems. However, the shortest path method may not lead to optimal results.

Nowadays, cloud computing is becoming prevalent. With the decrease of computing cost, cloud computing can be used to compute parallel algorithms with high efficiency. In traditional exact methods, the NP-hard programming is difficult to be decoupled into parallel independent subtasks. We found the Genetic Algorithm can be decoupled with some modifications. Therefore, in this paper, we propose a distributed parallel Genetic Algorithm that solves unsplittable link mapping of the VN embedding problem with high performance and low complexity.

We summarize the contributions of our proposed parallel GA algorithm as follows. 1) To the best of our knowledge, our work is the first to propose a GA algorithm for an online VN link mapping problem. Inspired by [7], our proposed algorithm deals with a more complex problem by solving GA formulated in matrix form. 2) Instead of generating one solution as previous research, we try to find the best solution among multiple feasible solutions by using genetic iterations. Extensive simulation results demonstrate that our approach provides better performances both on execution time and acceptance ratio than existing ones. 3) To solve the VN embedding problems efficiently, we propose a distributed parallel GA algorithm by applying variants in the GA. And the theoretical analysis shows the execution time can be reduced to logarithmic time. 4) Our proposed algorithm can be terminated with an intermediate solution at any time. If there is a time limitation for dynamic VN requests, parallel working machines can be added to increase execution speed to get an optimum for the GA algorithm. 5) Few papers discuss VN embedding with nonlinear objective functions in both exact methods and heuristic solutions. In this paper, we compare the performance between a linear objective function and a nonlinear one. Results show that the nonlinear objective function is more effective than the linear one.

The rest of this paper is organized as follows. Section II describes the network model and problem formulations. Section III introduces our proposed parallel GA algorithms. Section IV presents simulation results that compare the proposed algorithm with previous work. Finally, Section V concludes the paper.

II. Network Model

A. Substrate Network and VN Request

We consider a substrate network infrastructure $G^s = (N^s, E^s)$ composed of a set of substrate nodes N^s and a set of substrate bidirectional links denoted as E^s. Each substrate node $n^s \in N^s$ is associated with CPU capacity value $C(n^s)$ and its geographic location $loc(n^s)$. Each substrate link $e^s \in E^s$ has bandwidth capacity weight value $B(e^s)$. Every substrate node has a unique ID number that is used in genetic encoding as Fig. 1.

A VN request is modeled as a weighted graph, denoted by $G^v(t_a, t_d, D) = (N^v, E^v, t_a, t_d, D)$. N^v is a set of virtual nodes. E^v is a set of virtual links. t_a is arrival time of the VN request. t_d is the duration of the VN request and D represents the maximum distance between a virtual node and its associated substrate node. Each virtual node $n^v \in N^v$ in a VN request has CPU capacity requirement value $C(n^v)$ and a location $loc(n^v)$. The distance between a substrate node and the virtual node is denoted by $dis(loc(n^v), loc(n^s))$. Each link $e^v \in E^v$ has bandwidth requirement value $B(e^v)$. We also denote N as the number of virtual links.

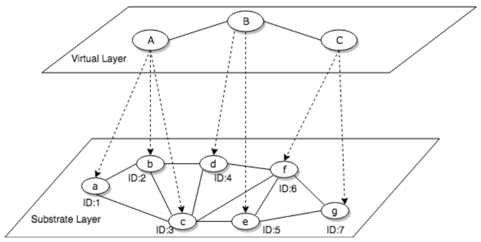


Fig. 1: An example of a virtual network with its associated substrate network.

B. VN Assignment

When a VN request arrives, the substrate network is supposed to decide if the request should be accepted on basis of current remaining resources. The assignment is released until the request departures.

1) Node Mapping: To allocate a VN request, we should map virtual nodes first. Each virtual node in a VN request should be allocated to a different substrate node. There are two constraints in node mapping:

$$C(n^v) \leqslant R^n(M(n^v)) \tag{1}$$

$$dis(loc(n^v), loc(M(n^v))) \leqslant D \tag{2}$$

where, $\quad M(n^v) \in N^s, \quad R^n(M(n^v)) \leqslant C(M(n^v))$

$M(n^v)$ is the substrate node mapping from a virtual node n^v. $R^n(M(n^v))$ is the remaining CPU capacity of the substrate node, which is updated after allocating/releasing each VN request.

2) Link Mapping: After node mapping, the VN embedding problem becomes multiple path embedding problems in the substrate layer. Each virtual link(unsplittable) is mapped to a substrate path(unsplittable) with one or more substrate links. We define a mapping $M(n^v, m^v)$ from a virtual link to a substrate path. Similar to $R^n(M(n^v))$, $R^e(e^s)$ is the remaining bandwidth of substrate link e^s. We define the set of all substrate paths from source node $M(n^v)$ to destination node $M(m^v)$ as $\mathcal{P}(M(n^v), M(m^v))$. Every substrate link in $M(e^v)$ should have enough bandwidth resource to allocate the virtual link:

$$B(e^v) \leqslant R^e(n^v, m^v) \tag{3}$$

where, $\quad \begin{aligned} e^v = (n^v, m^v), \quad M(e^v) \in \mathcal{P}^s(M(n^v), M(m^v)) \\ R^e(e^s) \leqslant B(e^s), \quad R^e(n^v, m^v) = \min_{e^s \in M(e^v)} R^e(e^s) \end{aligned}$

When a VN request mapping satisfies the constraints above for all nodes and links, it means the substrate network has enough resource to support the request by the mapping method. And the mapping is defined as a feasible solution.

C. Objectives

In our proposed link mapping algorithm, we try to balance the load for link mapping. The objective function used in link mapping is also called fitness function. The fitness function in this paper takes all links' usage as a whole, as opposed to other heuristic link mapping problems[1][2], which map virtual links sequentially. In addition, we try to measure the bandwidth cost of mapping the VN request as well as balance the load. The more remaining bandwidth left means more VN requests could be allocated in the future. Therefore, we encourage higher remaining bandwidth with smaller cost in the fitness function.

Existing solutions use linear objective functions to simplify optimization process. For example, the linear objective function in [1] is based on the remaining bandwidth of previous VN allocations as shown in (4). σ is a small positive constant to avoid the denominator becoming zero. f_{uv}^{i} describes the total amount of flow from u to v for the ith virtual link under the specific mapping scenario.

$$F_{LP} = \quad \sum_{e^s(u,v)\in E^S} \frac{\alpha_{uv}}{R^e(u,v)+\sigma} \sum_i f_{uv}^i \tag{4}$$

In our proposed algorithm, the linear or nonlinear function has almost similar computing complexity. We also formulate the fitness function as a nonlinear function as shown in (5), where the bandwidth usage of the current virtual link allocations becomes part of the denominator. A nonlinear fitness function like (5) improves the bandwidth efficiency since a small residual bandwidth in each substrate link can cause bandwidth fragmentation and is hard to be utilized for future VN requests in unsplittable link mapping. In the simulation, we keep both fitness functions and compare their performances.

$$F_{NLP} = \quad \sum_{e^s(u,v)\in E^S} \frac{\alpha_{uv}}{R^e(u,v)-\sum_i f_{uv}^i+\sigma} \tag{5}$$

III. Proposed VN Embedding Algorithm

With the development of computing capability, the cost of computing devices is decreasing. Therefore, people concern more on execution time rather than computing cost. Parallel method becomes prevalent to utilize more computing resources for the sake of time saving. Nevertheless, how to design the parallel structure for VN embedding link mapping is still a tough problem. Previous traditional algorithms focus on how to generate one feasible solution with good performance. However, parallelism in the process of seeking one good feasible solution is hard to realize. A feasible solution contains multiple virtual links that share the substrate link resources. Hence, virtual link mappings are highly dependent on each other, which makes parallelism complicated.

In contrast, there is no dependency among different feasible solutions because they are mutually exclusive. Genetic Algorithm is such an algorithm that applies naturally to parallel computing. In general, GA seeks the best solution by evaluating and improving multiple feasible solutions through an evolution process. We propose a parallel GA method that can be run over many machines in a distributed way. Each chromosome denotes a feasible solution. Each parallel working machine can run independently and generate descendant chromosomes. After all required iterations have been executed by each machine independently, the best feasible solution will be selected among all parallel machines.

The proposed parallel structure is shown in Fig. 2. Procedures such as node mapping and synchronization work are still working sequentially. We call the procedures running sequentially as master working procedures. And the slave working procedures indicate the procedures running independently in slave nodes.

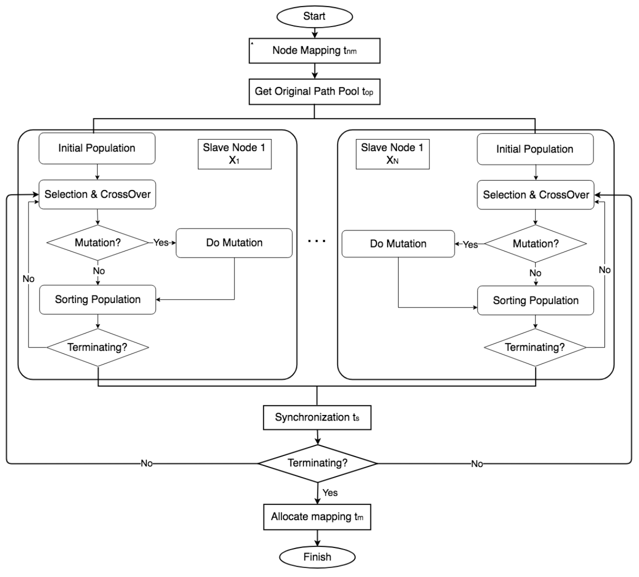


Fig. 2: Parallel execution flow chart

A. Node Mapping Algorithm

Node mapping algorithm in this paper is a greedy method instead of an exact formulation since an exact method is always computationally expensive but also may not lead to a better solution. To achieve efficiency and expense reduction, we take a similar heuristic algorithm as [2].

$Y(n^s)$ shown in (6) calculates the remaining substrate node resources. Specifically, $Y(n^s)$ is not only dependent on the remaining CPU capacity $R^n(n^s)$. Moreover, $Y(n^s)$ considers remaining link resources of all adjacent substrate links. $E(n^s)$ is the set of adjacent substrate links of node n^s. σ_1 and σ_2 denote weights to control the significance of $R^n(n^s)$ and $R^e(e^s)$, which should be non-negative.

$$Y(n^s) = \sigma_1 \times R^n(n^s) + \sigma_2 \times \sum_{e^s\in E(n^s)} R^e(e^s) \tag{6}$$

B. Genetic Representation

After node mapping, a VN request problem becomes multiple link mapping problems. A chromosome $\boldsymbol{c_i}$ denoted by (7) represents a feasible solution for mapping a VN to a substrate network. The subscript i indicates the ith element in GA population. A gene $\boldsymbol{g_{ij}}$ is a substrate path corresponding to a virtual link and has two subscripts. The first subscript i indicates its chromosome, and the second subscript j denotes the jth virtual link in the chromosome. Similarly, A node denoted by n_{ijk} represents a substrate node ID with three subscripts. The first two subscripts indicate its gene, and the third one denotes its position in the gene. A gene can be denoted by (8) with a variable length d_{ij}. Each gene $\boldsymbol{g_{ij}}$ can be divided into two partial paths as (9): head $\boldsymbol{H_{ijk}}$ and tail $\boldsymbol{T_{ijk}}$, where k indicates the index of node in the gene.

$$\boldsymbol{c_i} = \{\boldsymbol{g_{i1}}, \boldsymbol{g_{i2}}, ..., \boldsymbol{g_{ij}}, ..., \boldsymbol{g_{iN}}\} \tag{7}$$

$$\boldsymbol{g_{ij}} = \{n_{ij1}, ..., n_{ijk}, ..., n_{ijd_{ij}}\} \tag{8}$$

$$\boldsymbol{g_{ij}} = \left[\boldsymbol{H_{ijk}}, \boldsymbol{T_{ijk}}\right], \ \ \forall k \in (0, d_{ij}) \tag{9}$$

where, $$\boldsymbol{H_{ijk}} = \left[n_{ij1}, n_{ij2}, \ldots, n_{ijk}\right]$$

$$\boldsymbol{T_{ijk}} = \left[n_{ij(k+1)}, n_{ij(k+2)}, \ldots, n_{ijd_{ij}}\right]$$

C. Link Mapping Algirithm

1) Getting Original Path Pool: Before we conduct our link mapping process, we need to find some good potential paths in the substrate network for mapping virtual links. To this end, shortest paths based on hop count are certainly more favorable because they tend to consume fewer resources. For each source-destination pair in the substrate network, we identify K shortest paths as our path pool. Existing K shortest path algorithm can be readily deployed to build the path pool. This process can be done before any online VN requests arrive. Therefore, we do not count the time for this process as part of the time for our online embedding process. Each virtual link has K substrate static paths, hence the original path pool should have $K \times N$ paths.

2) Slave Working Procedure: In this procedure, each slave node performs the GA and gets an independent solution.

a) Initialization: The purpose of the initialization is to generate a population of M chromosomes, where each chromosome is a feasible solution for embedding a VN. To select a chromosome, we need to select genes that form a feasible solution. This is done in two steps. The first step is to choose a candidate gene by uniformly selecting a path from the K shortest paths associated with the mapping of a specific virtual link in the path pool. We use uniform selection instead of another specific order since our proposed algorithm focuses on parallel running.

After all candidate genes in a chromosome have been selected, the feasibility of the chromosome should be checked as described in Section II-B. Then the feasible chromosome will be added to the initial population denoted by $\boldsymbol{P}$. If the candidate chromosome does not pass the feasibility test, we will go back to step 1 to select another candidate and test again. This process continues until a feasible chromosome is selected. $\boldsymbol{P}$ can be described as a matrix shown in Equation (10). Generally, after initialization, the population $\boldsymbol{P}$ should have M chromosomes and each chromosome has N virtual links. Therefore, the size of $\boldsymbol{P}$ is $M \times N$.

$$\boldsymbol{P} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_i \\ \vdots \\ c_M \end{bmatrix} = \begin{bmatrix} g_{11} & \cdots & g_{1j} & \cdots & g_{1N} \\ g_{21} & \cdots & g_{2j} & \cdots & g_{2N} \\ \vdots & \ddots & & \ddots & \vdots \\ g_{i1} & \cdots & g_{ij} & \cdots & g_{i} \\ \vdots & \ddots & & \ddots & \vdots \\ g_{M1} & \cdots & g_{Mj} & \cdots & g_{MN} \end{bmatrix} \tag{10}$$

b) Selection: Selection operation is to select parent chromosomes from the initial population for the crossover operation. In general, one or several pairs of parent chromosomes can be chosen from this step. In this paper, we aim to improve the degree of parallelism, we only choose one-pair parents. We perform the selection scheme based on random selection with replacement. In the natural world, crossover in close relatives should be avoided. In general, the parents' chromosomes should be removed from the population after crossover operation. However, in this paper, crossover between a good parent and a good child may produce a better result with high probability.

c) Crossover: In crossover and mutation, the operations are based on genes. We denote two parent chromosomes as $\boldsymbol{c_s}$ and $\boldsymbol{c_r}$. s and r indicate the chromosome's index in current population. We denote the two new children chromosomes from crossover operation as $\boldsymbol{c_{(M+1)}}$ and $\boldsymbol{c_{(M+2)}}$. The genes inside parent chromosomes can be denoted as $\boldsymbol{g_{sj}}$ and $\boldsymbol{g_{rj}}$. Each gene in the chromosome should perform crossover with the counterpart of the other parent chromosome. For each pair of genes, we first identify a common node. If there is a node n_{sju} in $\boldsymbol{g_{sj}}$ equal to a node n_{rjv} in $\boldsymbol{g_{rj}}$, where u and v are not the indexes of source or destination node, we denote the node as a common node.

When the common node is selected in two genes, the common node is utilized as an intermediate node. With the intermediate node as the demarcation point, we swap the second parts of the two genes to generate two children genes. It is easy to see, children generated in such a way are still valid paths. The children genes are defined as (11) and (12).

$$g_{(M+1)j} = \begin{cases} \left[\boldsymbol{H_{sju}}, \boldsymbol{T_{rjv}}\right] & \text{for } p \in (0, 0.5] \quad \text{(11a)} \\ \left[\boldsymbol{H_{rjv}}, \boldsymbol{T_{sju}}\right] & \text{for } p \in (0.5, 1] \quad \text{(11b)} \end{cases}$$

$$g_{(M+2)j} = \begin{cases} \left[\boldsymbol{H_{rjv}}, \boldsymbol{T_{sju}}\right] & \text{for } p \in (0, 0.5] \quad \text{(12a)} \\ \left[\boldsymbol{H_{sju}}, \boldsymbol{T_{rjv}}\right] & \text{for } p \in (0.5, 1] \quad \text{(12b)} \end{cases}$$

Each pair of parent genes can produce two children genes, hence there exist 2^N gene combinations for a child chromosome. However, we only pick two chromosomes based on uniform possibility defined as p in (11) and (12) regardless of their fitness values. There are two reasons for the random selection. The new chromosome with good fitness value generated in crossover may become inferior after mutation. Therefore, a specific selection criteria in the crossover procedure may not work well. The other reason is based on the particularity of a path chromosome. In our GA, when the children chromosomes generate their next generation, they may create the chromosomes same as their parents or ancestors. If we always choose the fittest children, the Genetic Algorithm may be trapped in loops, and also produce high repetitive rate especially for a parallel Genetic Algorithm. After two children chromosomes have been generated, the population now can be represented by (13) that adds two children chromosomes.

$$\boldsymbol{P} = \begin{bmatrix} c_1 \\ \vdots \\ c_s \\ \vdots \\ c_r \\ \vdots \\ c_M \\ c_{M+1} \\ c_{M+2} \end{bmatrix} = \begin{bmatrix} g_{11} & \cdots & g_{1j} & \cdots & g_{1N} \\ \vdots & \ddots & & \ddots & \vdots \\ g_{s1} & \cdots & g_{sj} & \cdots & g_{sN} \\ \vdots & \ddots & & \ddots & \vdots \\ g_{r1} & \cdots & g_{rj} & \cdots & g_{rN} \\ \vdots & \ddots & & \ddots & \vdots \\ g_{M1} & \cdots & g_{Mj} & \cdots & g_{MN} \\ g_{(M+1)1} & \cdots & g_{(M+1)j} & \cdots & g_{(M+1)N} \\ g_{(M+2)1} & \cdots & g_{(M+2)j} & \cdots & g_{(M+2)N} \end{bmatrix} \tag{13}$$

In this crossover scheme, if there are two or more than two common nodes existing, the cross point will be chosen randomly with uniform possibility. If there is no common node found, the outputs from crossover are supposed to keep the parent genes to ensure the whole procedure running normally.

d) Mutation: After crossover, every gene of a child chromosome undergoes mutation operation with a fixed probability(mutation rate). If a gene is selected to perform mutation operation, it replaces the partial route between two nodes with the shortest path chosen from the path pool to connect the two mutation nodes. The mutation nodes which are selected randomly with uniform possibility inside the gene. As shown in (14), if there is no mutation operation, the gene should be kept for the next step. If mutation occurs, the alternative path $\boldsymbol{P_{wz}}$ between mutation node $n_{ij(w+1)}$ and n_{ijz} will update current gene. We use g'_{ij} denoting the child gene after mutation.

$$\boldsymbol{g'_{ij}} = \begin{cases} \boldsymbol{g_{ij}} & \text{if no mutation} \quad (14a) \\ [\boldsymbol{H_{ijw}}, \boldsymbol{P_{wz}}, \boldsymbol{T_{ijz}}] & \text{if mutation} \quad (14b) \end{cases}$$

e) Sorting Population and Termination Conditions: After the new chromosomes have been generated from the crossover/mutation operation, they have to go through the feasibility check. Occasionally, there may exist loops inside genes[7]. Therefore, refinement check is necessary to detect and remove loops inside genes. The chromosome that fails the feasibility check will be removed from the population. If both two new chromosomes fail the feasibility check, we will go back to crossover operation to generate two different chromosomes until a feasible solution found. This procedure will be stopped when the maximum count is reached or there are no different children chromosomes available.

The population is supposed to be re-sorted by the fitness value after the feasibility and refinement check. Only the best M(population size) chromosomes are saved to the next generation. Generally, the GA algorithm will be terminated when the best chromosome of the population has not been changed t times in succession. t is a tuning parameter called terminating parameter. The GA algorithm also can stop when a fixed number of generations reached.

3) Synchronization and Allocating the VN Request: After slave nodes finish parallel procedures, all population generated by slave nodes should be resorted by the fitness value in the master node. The best chromosome becomes the final solution for link mapping. Then the VN request should be allocated into the substrate network. After that, the substrate network is supposed to update its residual resources.

4) Execution time of parallel GA: As shown in Fig. 2, we define execution time of each procedure as t_{nm} for node mapping, t_{op} for generating original path pool, X_i for the ith slave working procedure, t_s for Synchronization and t_m for allocating the VN request.

Slave procedures can be executed independently in parallel, hence the total time for slave procedures is depended on the slowest one as shown in (15). We define n as the parallel level. The n can be tuned according to the tradeoff between available computing resource and expected finishing time. When we evaluate the time saving of our distributed parallel structure, the time of sequential running for all slave working procedures is required as a criterion. Equation (16) shows the time consuming of sequential slave working procedures.

$$Y_n = max\{X_1, X_2, ..., X_n\} \tag{15}$$

$$S_n = X_1 + X_2 + ... + X_n \tag{16}$$

Master procedures have to be performed sequentially. Therefore, the total execution time to embed a VN request can be evaluated by (17) for parallel slave running or (18) for sequential slave running.

$$T_p = t_{nm} + t_{op} + Y_n + t_s + t_m \tag{17}$$

$$T_s = t_{nm} + t_{op} + S_n + t_s + t_m \tag{18}$$

X_i can be considered as a continuous random variable, which can be formulated with a continuous probability distribution. Since all parallel nodes work independently and perform the same GA procedure, all the execution time of parallel working nodes can be considered as independent-identically-distributed variables. According to the numerical results in Section IV, the histogram distribution of X_i can be acquired in Fig. 3, which obeys Inverse Gaussian distribution.

For sequential slave process running, the mean time of total slave procedures S_n is supposed to increase linearly over n. The expectation of S_n is easily evaluated as (19).

$$\mathbb{E}[S_n] = n\mathbb{E}[X_i] \tag{19}$$

To evaluate the upper bound of Y_n's mean value, Cramer-Chernoff method is applied, which makes use of the Moment Generating Function of X_i. An Inverse Gaussian random variable X with parameters μ and λ has Moment Generating Function shown in (20).

$$M(t) = \mathbb{E}[e^{t \cdot X}] = e^{(\frac{\mu}{\lambda}(1-\sqrt{1-\frac{2\mu^2 t}{\lambda}}))} \quad \text{for } \mu, \lambda > 0 \tag{20}$$

By Jensen's inequality,

$$\begin{aligned} e^{t\mathbb{E}[Y_n]} &\leq \mathbb{E}[e^{t \cdot Y_n}] = \mathbb{E}[\max_i\{e^{t \cdot X_i}\}] \\ &\leq \sum_{i=1}^{n} \mathbb{E}[e^{t \cdot X_i}] = n\mathbb{E}[e^{t \cdot X_i}] \\ \implies \mathbb{E}[Y_n] &\leq \frac{log(nM(t))}{t} = \frac{log(n)}{t} + \frac{\frac{\mu}{\lambda}(1-\sqrt{1-\frac{2\mu^2 t}{\lambda}})}{t} \end{aligned} \tag{21}$$

Since t, λ and μ are constant values, we can conclude that the increasing tendency of parallel running is logarithmic. Compared with the linearity of sequential running, our proposed distributed parallel algorithm saves much more time when the parallel level is larger.

IV. Performance Evaluation

A. Comparison Method

In this paper, we compare our GA-R and GA-LP algorithms with three other algorithms. GA-R indicates the random parallel GA using NLP fitness function, while GA-LP represents the random parallel GA using LP fitness function. We use the same simulation setting as previous research[1]. Specifically, we select SP[2] because it is considered the fastest algorithm due to its simplicity. R-ViNE and D-ViNE[1] are se.lected because they tend to have the best performance and are typically used as benchmark for comparison. We choose unsplittable link mapping in R-ViNE and D-ViNE.

B. Evaluation Results

There are several performance metrics for evaluation purposes in this simulation. We measure the average acceptance ratio and average remaining bandwidth of the substrate network.

1) Time and parallel level analysis: Time analysis is simulated only on GA-R since GA-R and GA-LP have a similar computation complexity. we simulate the GA-R using one machine that all parallel working node sequentially. For the simulation of parallel GA, we implement our algorithm in one machine and the parallel processes are executed sequentially.

We collected numerous execution time of parallel working processes, and tried to find the time distribution. As shown in Fig. 3, X_i fits Inverse Gaussian distribution with the sum of square error $4.19e^{-5}$.

As described in (17), the total execution time T_p can also be estimated based on sequential running results. We perform the simulation with different parallel level from 4 to 32. From Fig. 4a, we can inspect that acceptance ratio increases by the growth of parallel level and finally acceptance ratio converge to limit value. Fig. 4b gives the average total execution time for one VN request by using different algorithms, where our proposed algorithm has a parallel level equals 16.

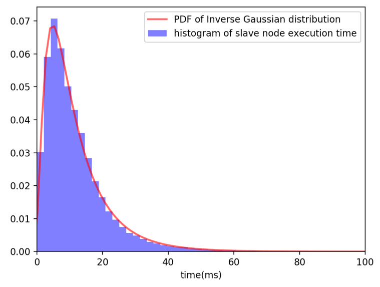
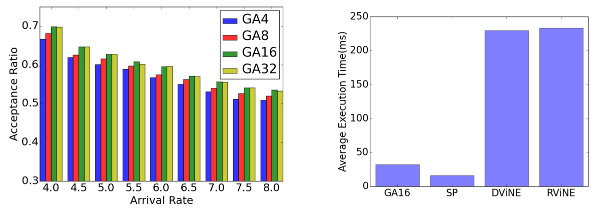


Fig. 3: Time distribution of X_i

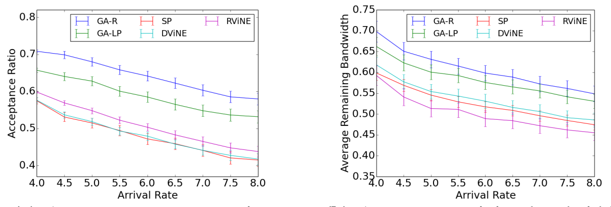
2) Performance Results: The parallel level of both GA-R and GA-LP is 16 since the performance of GA-R is getting converged after $n = 16$ as shown in Fig. 4a. Figure 4b shows that our algorithm with parallel level of 16 can complete the task in nearly the same time as the fastest algorithm SP. The figures in Fig. 5 show the average values over different arrival rates from 4 to 8 per 100 time units with 95% CI(confidence interval). These results are generated by averaging over two different substrate networks with same parameters (described in simulation settings).

Our algorithms GA-R and GA-LP have more acceptance ratio with less bandwidth cost as shown in Fig. 5. The gain comes from more feasible solutions evaluated in our proposed algorithms. As described before, compared heuristic algorithms only try to guess one feasible solution for the virtual link mapping. Instead of taking time to generate one greedy solution, our algorithms utilize parallel running with GA to consider more feasible solutions with relatively small time consuming. Besides, we can conclude that our proposed algorithm GA-R using nonlinear fitness function has better performance than the proposed algorithm GA-LP using linear fitness. Specifically, GA-R has more acceptance ratio, higher remaining bandwidth than GA-LP.



(a) Average acceptance ratio over arrival rates **(b)** CPU execution time over different algorithms

Fig. 4: Acceptance and execution time performance



(a) Average acceptance ratio **(b)** Average remaining bandwidth

Fig. 5: Performance comparison over arrival rates

V. Conclusion

Existing research in VN embedding problems focuses on either scalability or optimality. In this paper, we take both scalability and optimality into account. We proposed a distributed parallel Genetic Algorithm that has never been used in online VN link Embedding. The simulation shows that our GA-R algorithm has higher bandwidth efficiency and better performance than other previous research. Execution time, as well as performance on different parallel levels, are also analyzed in this paper. We evaluate the execution time model and demonstrate that the execution time complexity is decreased to $O(log(n))$ comparing with sequential running $O(n)$. Besides, we also introduce a nonlinear fitness function, which has been shown to improve the link mapping efficiency. In future work, we will look into feasible ways to support splittable link mapping. If splittable link mapping is indeed feasible in real networks, we will further study GA based splittable link mapping algorithms as our future task.

References

[1] Mosharaf Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Vineyard: Virtual network embedding algorithms with coordinated node and link mapping. *IEEE/ACM Transactions on Networking (TON)*, 20(1):206–219, 2012.

[2] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, 2008.

[3] Zibo Zhou, Xiaolin Chang, Yang Yang, and Lin Li. Resource-aware virtual network parallel embedding based on genetic algorithm. In *2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 81–86. IEEE, 2016.

[4] Peiying Zhang, Haipeng Yao, Maozhen Li, and Yunjie Liu. Virtual network embedding based on modified genetic algorithm. *Peer-to-Peer Networking and Applications*, pages 1–12, 2017.

[5] Changcheng Huang and Jiafeng Zhu. modeling service applications for optimal parallel embedding. *IEEE Transactions on Cloud Computing*, 2016.

[6] Georgios S Paschos, Mohammed Amin Abdullah, and Spyridon Vassilaras. Network slicing with splittable flows is hard. In *2018 IEEE 29th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, pages 1788–1793. IEEE, 2018.

[7] Chang Wook Ahn and Rudrapatna S Ramakrishna. A genetic algorithm for shortest path routing problem and the sizing of populations. *IEEE transactions on evolutionary computation*, 6(6):566–579, 2002.