Scheduling Jobs across Geo-Distributed Data Centers

A Genetic Algorithm Based Approach

Ziyin Zhang (daenerystargaryen@sjtu.edu.cn),
Zihang Xu (hang123@sjtu.edu.cn),
Hanqing Huang (inivia@sjtu.edu.cn)

Department of Computer Science,
Shanghai Jiao Tong University, Shanghai, China

Abstract. With the popularization of distributed systems and exponential growth of data volumes, it is becoming ever more impractical - technologically, economically, or politically - for a data analytic job to gather all the data it needs to one site and then process them locally. In face of this challenge, computation migration was born, and with it the problem of how to schedule jobs across geo-distributed data centers efficiently. In this paper, we will first use Travelling Sales Man, or TSP, a typical NP-complete problem, to prove the NP-hardness of Distributed Job Scheduling, and then propose a genetic algorithm based approach to tackle this problem.
Keywords: distributed job scheduling, genetic algorithm.

1 Introduction

With the growing trend of distributed systems and concomitant data decentralization, scheduling jobs across geo-distributed data centers has become a major task faced by business managers as well as computer scientists. Sometimes it is no longer economical, or even possible, to use the conventional method when scheduling processes in these systems, where data are transferred to the site of the requesting process. Instead, migrating computation to where the data resides, and transfer the result back is a much more feasible solution for data-intensive jobs.

However, this trend poses new challenges as well. Scheduling jobs across a distributed system is far more difficult than the traditional process scheduling inside a single computer [4], as the transfer time of data counts for a major part of the processing overhead. And with the amount of data each job needs and the distance they need to travel differing dramatically, sometimes it could be hard to guarantee fairness between different jobs as well. Some conventional approaches to solving this problem include but are not limited to Shortest Remaining Processing Time based scheduling, Reording baced approach, and Workload-aware approach [2]. However, these approaches either fail to approximate optimality, or are too mathematically sophisticated for college students to understand. In this paper, we use a different, genetic based, and much more simple algorithm to address this problem. It arguably does not always yield the optimal solution, but is nonetheless quite efficient in practice.

2 Problem Formalization

2.1 Basic Assumptions

In the following part of this paper, the words job, process, and task are used interchangeably, as are location, data center, site; edge, link; and computation slot, processor.

Suppose we have k distributed data centers, each with one or several computation slots. For simplicity, we assume that all processors are homogeneous. There are m jobs waiting to be allocated to processors, where they can gather the necessary data and execute. And there are totally n types of data sources, some of which resides in a certain data center, while some other are intermediate results produced by analytical jobs.

Since data privacy is also an important factor to consider when scheduling jobs across distributed systems, we also assume that the data sent from one site to a certain process is exclusive to that process, i.e. other processes at the same site can not use this data. If they wish so, their own copy of data needs to be sent from the source.

2.2 Notations

Now we will define the notations used in this paper. Note that capitalized italic letters stand for sets, while capital blackboard letters stand for collections of sets, and bold small letters stand for vectors or matrices.

- Set of locations $DC = \{DC_1, DC_2, \cdots, DC_k\}$
- Number of slots in each site $\mathbf{s}[k] = (s_1, s_2, \cdots, s_k)$, where DC_i has s_i computation slots.
- The locations of each data source $\mathbf{ds}[n] = (ds_1, ds_2, \cdots, ds_n)$. $ds[i] = j$ denotes that data i resides in site j. Initially, the locations of certain types of data may be -1, indicating that they are intermediate data. After some processes produce them, their locations will be modified accordingly.
- Data demand matrix $\mathbf{d}[m][n] = (d_{i,j})_{m\times n}$, where $d_{i,j}$ denotes that process i needs $d_{i,j}$ MB of data n. Vector $\mathbf{d}_i$ denotes all the data task i needs.
- The set of all demands D.
- The set of edges E, i.e. the directed links between data centers.
- The data that each process produces $\mathbf{dataproduced}[m]$. $\mathbf{dataproduced}[i] = j$ denotes that process i produces data j. If $j = 0$, it means process i produces no data.
- Link matrix $\mathbf{c}[k][k] = (c_{i,j})_{k\times k}$. The bandwidth of the data path from site i to site j is $c_{i,j}$. And if $c_{i,j} = 0$, it means that there is no data path from site i to j. Also note that $c_{i,i} \neq \infty$, which means data transfer in the same site has limits as well, although much larger than that of inter-site transfers.
- Execution time of each task $\mathbf{t}[m] = (t_1, t_2, \cdots, t_m)$.
- Task allocation vector $\mathbf{a}[m] = (a_1, \cdots, a_m)$ denotes where each task is allocated. $a_i \in DC, \forall i \in [1, m]$
- Paths $\mathbb{P}$. $\mathbb{P}$. Each element P_{ij} in it is a set of (one or several) edges, which denotes the transfer path of data j from its source to the process i demanding it. Each $P_{ij} \in \mathbb{P}$ should starts from $\mathbf{ds}[j]$, and ends at $\mathbf{a}[i]$.
- Bandwidth allocation vector $\mathbf{x}$, with its size equal to $|\mathbb{P}|$. Each element x_{ij} in it is the bandwidth allocated to path P_{ij}. (Note that this vector is not fixed, but changes dynamically with time, as some processes start executing or terminate.)
- Start time $\mathbf{st}[m] = (st_1, \cdots, st_m)$, and finish time $\mathbf{ft}[m] = (ft_1, \cdots, ft_m)$.

Remark 1. $DC, \mathbf{s}, \mathbf{ds}, \mathbf{d}, \mathbf{dataproduced}, \mathbf{c}, \mathbf{t}$ are given with the task requests. $\mathbf{a}, \mathbb{P}, \mathbf{x}, \mathbf{st}, \mathbf{ft}$ are left for the algorithm designers to specify.

Remark 2. Once $\mathbf{a}$ is determined, $(\forall i)((\exists j)\mathbf{dataproduced}[j] = i \rightarrow \mathbf{ds}[i] = \mathbf{a}[j])$, i.e. the sources of all the data, including those produced by processes, are defined as well.

Remark 3. A job will start executing when all the data it needs are ready. So $st_i = \max_j\{ft_j + l \times d_{i,j}/x_{d_{i,j}}\}$, where l denotes the length of $P_{d_{i,j}}$(the number of edges it passes), and ft_j denotes the time the process which produces data j is finished. If data j is not involved in precedence constraints, $ft_j = 0$. And $ft_i = st_i + t_i$ for all jobs.

2.3 Objectives

With all the notations we will use clarified, we define the concept of feasible solutions.

Definition 1. *An allocation vector* $\mathbf{a}$ *is feasible, if and only if for all the demands* $d_{ij} \in D$*, there exists a path from* $\mathbf{ds}[j]$ *to* $\mathbf{a}[i]$*, and* $\sum_{a_i\in\mathbf{a},a_i=j} 1 \leqslant \mathbf{s}[j]$, $\forall j \in [1, k]$.

This definition basically guarantees that there exists a feasible path for every data demand, and that the number of processes allocated to each site does not exceed its capacity.

Definition 2. *A bandwidth vector* $\mathbf{x}$ *is feasible, if and only if for every edge* $e \in E$, $\sum_{e\in P,P\in\mathbb{P}} x_P \leqslant c_e$

Definition 3. *With a given job allocation* $\mathbf{a}$*, a bandwidth allocation* $\mathbf{x}$ *achieves max-min fairness when for any demand* $d \in D$ *such that in any* $\mathbf{x}'$ *with* $x'_d > x_d$*, there exists a demand* d' *such that* $x'_{d'} < x_{d'}$ *[1].*

The expression of max-min fairness is not that straightforward, but the principle behind it is intuitive: all the paths passing a particular edge at a particular time should share the same portion of the edge's bandwidth, unless an edge doesn't need that much.

What we need to do is to find the optimal $\mathbf{a}$ among all the feasible allocations that, with a proper allocation of paths and bandwidths $\mathbb{P}$ and $\mathbf{x}$, yields the minimal average finish time $\frac{1}{m}\sum_{i=1}^{m} ft_i$, under the condition that max-min fairness is achieved throughout the execution of tasks.

2.4 Example

In this paper, we will mainly use an example of 13 data centers, 31 computation slots and 27 tasks to illustrate how our algorithm works [1]. The computation slots are distributed as shown in table 1. And the data locations and link bandwidths are shown in table 2, table 3 respectively. The demands are too many to be incorporated here, please refer to [1] upon demand.

Table 1. Number of processors in each site.

DC1	DC2	DC3	DC4	DC5	DC6	DC7	DC8	DC9	DC10	DC11	DC12	DC13
2	1	3	2	1	5	2	2	2	1	2	4	4

Table 2. Locations of existing data.

A1	A2	B1	B2	C1	C2	D1	D2	D3	E1	E2	E3	E4	F1	F2	F3	F4	F5
DC1	DC4	DC1	DC3	DC2	DC6	DC5	DC7	DC9	DC4	DC6	DC8	DC11	DC9	DC13	DC10	DC12	DC13

Table 3. The bandwidths of links between sites, in terms of MBps.

	DC1	DC2	DC3	DC4	DC5	DC6	DC7	DC8	DC9	DC10	DC11	DC12	DC13
DC1	1200	80	150	-	-	-	-	-	-	-	-	500	-
DC2	100	1200	120	160	200	-	-	-	-	-	-	-	-
DC3	80	100	1200	200	50	-	-	300	-	-	400	-	-
DC4	-	150	180	1200	20	120	-	-	200	-	-	-	-
DC5	-	200	40	20	1200	150	200	-	-	-	-	-	-
DC6	-	-	-	250	180	1200	-	90	-	-	50	-	500
DC7	-	-	-	-	200	-	1200	-	-	70	40	-	-
DC8	-	-	300	-	-	60	-	1200	20	90	-	-	-
DC9	-	-	-	300	-	-	-	30	1200	-	90	500	-
DC10	-	-	-	-	-	-	70	50	-	1200	-	-	-
DC11	-	-	400	-	-	40	60	-	50	-	1200	-	-
DC12	500	-	-	-	-	-	-	-	400	-	-	1200	500
DC13	-	-	-	-	-	400	-	-	-	-	-	500	1200

3 Problem Complexity

It is putatively considered that Distributed Job Scheduling is a NP-hard problem [2,3]. Failing to find a way to prove that it belongs to the NP class either, we will show here that Distributed Job Scheduling is polynomial reducible from Travelling Salesman Problem, or TSP, a classical NP-complete problem.

First we specify the decision problem of Distributed Job Scheduling: is there a feasible allocation of the jobs, such that the average completion time of all jobs is $\leqslant t_0$ under guaranteed max-min fairness?

Given a set of n cities $u_1, u_2, \cdots, u_n$ and pairwise distance function $d(u_i, u_j)$, we construct a Distributed Job Scheduling problem: Let there be n fully connected locations $v_1, v_2, \cdots, v_n$, each with one computation slot, where v_i maps u_i in TSP. Both c_{ij} and c_{ji}, i.e. the bandwidths between v_i and v_j in both directions, are $\frac{1}{d(u_i,u_j)}$ (in terms of MB/s). And there is an additional site v_{n+1}. $\forall i \in [1, n], c_{i(n+1)} = d_m, c_{(n+1)i} = 0$, where $d_m = \max\{d(u_i, u_j) | 1 \leqslant i, j \leqslant n, i \neq j\}$ in the TSP.

There are n tasks $t_1, t_2, \cdots, t_n, t_{n+1}$, and the intermediate result produced by t_i is needed by t_{i+1} in a quantity of 1 MB, for all $1 \leqslant i \leqslant n-1$. t_1 doesn't need any data source, while t_{n+1} needs 1 MB of the data produced by all other n tasks. There are no other data sources required by the rest tasks. The execution time of each task is 1 second.

Observation There exists an optimal scheduling with no time slack, i.e. every task starts executing the moment the data it needs are all gathered.

Proof. Otherwise, move the tasks with a time slack forward in time, and their finishing time, along with the finishing time of those tasks that come after them, will not increase.

Lemma 1. *There is a tour of length $\leqslant D$ for the salesman, if and only if there is a scheduling scheme with an average completion time of $\leqslant \frac{D}{n} + 1$ in terms of seconds.*

Proof. Firstly, if there is a tour $u_1, u_2, \cdots, u_n, u_1$ of length $\leqslant D$ for the salesman, allocate t_i to the corresponding site v_i. Since for all $i > 1$, t_i can not start until t_{i-1} finishes execution, the finish time of t_i is exactly the time our salesman arrives at c_i, if we temporarily assume that tasks finish executing immediately after they start. Under this assumption, the average finishing time is $\leqslant D/n$. When execution time is taken into consideration, the average finishing time is $\leqslant D/n + 1$ since all the tasks are executed in sequence and have the same execution time.

Secondly, suppose there is a scheduling scheme where t_i is allocated to c_i with an average completion time of $\leqslant \frac{D}{n} + 1$. Similarly, after ignoring the execution time, the average completion time is $\leqslant \frac{D}{n}$ because all the tasks must execute in sequence $(t_1, t_2, \cdots, t_n)$.

From the observation above, it can be concluded that the total time used is $\leqslant D$, since all the tasks execute in sequence. Thus if a salesman travels the cities in the same order as $(u_1, u_2, \cdots, u_n)$ and then back to u_1, the total distance is $\leqslant D$.

From Lemma 1, we have the following therom.

Theorem 1. TSP $\leq_p$ Distributed Job Scheduling.

And it can be concluded that Distributed Job Scheduling is a NP-hard problem.

4 A Genetic Algorithm Based Approach

4.1 Motivation

Given the NP-hardness of Distributed Job Scheduling, it may be impossible to find an efficient algorithm that yields the exact optimal solution, and close approximations that are efficient in practice should suffice. Considering the similarity between Distributed Job Scheduling and TSP, it naturally follows that we can address this problem with genetic algorithm, which, in the authors' past experience, performs outstandingly well in solving TSP.

Admittedly, genetic algorithm does not always yields the optimal solution, but nonetheless with carefully tuned genetic parameters achieves a satisfactory approximation. Still, the most obvious advantage of genetic algorithm is its simplicity and wide applicability. It can be used to tackle almost all NP problems, so long as a genetic coding scheme can be found.

And in this paper we will simply use the permutation of all the computation slots in different data centers put together as the genetic code, to guarantee uniformity when the tasks are randomly distributed. This permutation of slots will be mapped to an allocation of tasks in algorithm 5.

4.2 Evolution

Algorithm 1 is the top module of our genetic algorithm. The genetic parameters are defined in the first line, among which *sample* is the size of the population; G is the number of generations; *selected* is the number of individuals selected in each iteration; and pc, pm, ps stand for the probabilities of crossing, mutation and survival respectively. These parameters are empirically set to 200, 70, 10, 0.8, 0.35, 0.1 respectively in this example.

For reference, note that the following pseudo code in this paper follows the style of MATLAB for the most part.

Algorithm 1: genetic algorithm

parameters :$sample, G, selected, pc, pm, ps$;
$ga = \text{zeros}(sample, slots + 1)$;
for $i = 1$ *to* $sample$ **do**
  $ga(i, 1 : slots) = \text{randperm}(slots)$;
end
for $i = 1$ *to* $sample$ **do**
  $ga(i) = \text{DJS\_time}(ga(i))$;
  while $ga(i, slots + 1) = 0$ **do**
    $ga(i, slots) = \text{randperm}(slots)$;
    $ga(i) = \text{DJS\_time}(ga(i))$;
  end
end
for $gen = 1$ *to* G **do**
  $ga = \text{DJS\_select}(ga, selected, ps)$;
  if $rand() \leq pc$ **then**
    $ga = \text{DJS\_cross}(ga)$;
  end
  $ga = \text{DJS\_mutate}(ga, pm)$;
end
$[opt\_avg\_finish\_time, index] = \min(ga(:, slots + 1))$;
$\text{DJS\_time}(ga(index))$;

Algorithm 1 defines a matrix of size $sample \times (slots + 1)$ to record the information of the population, where $slots$ is the number of total slots in all the data centers, and the last element of each row is used to store the time information.

Initially the population is randomly generated, and DJS_time is called to calculate the fitness (in this example, average finish time of all the tasks) of these individuals. If the fitness is 0 (which means that the sample is not a feasible solution), the sample is replaced immediately. And then in each iteration - or generation - the processes selecting, crossing and mutating are executed with the predefined probabilities.

After the main part of the genetic algorithm is finished, the minimal average finish time is found from the last column of the matrix, and DJS_time is called again to find the allocation information.

Algorithm 2: DJS_mutate

Input: ga, pm
Output: ga
$ga' = ga$;
$flag = \text{zeros}(1, sample)$;
for $i = 1$ *to* $sample$ **do**
  if $rand() \leq pm$ **then**
    Randomly choose two points x_1, x_2 in $ga(i)$;
    Reverse the genetic code between x_1, x_2;
    $flag(i) = 1$
  end
end
for $i = 1$ *to* $sample$ **do**
  if $flag(i)$ **then**
    $ga(i) = \text{DJS\_time}(ga(i))$;
    if $ga'(i, slots + 1) < ga(i, slots + 1)$ *or* $ga(i, slots + 1) = 0$ **then**
      $ga(i) = ga'(i)$;
    end
  end
end

As is shown in algorithm 2, the mutating procedure uses a simple reversing method to create new features that will possibly help the population jump out of the locally optimal solution. It then compares the mutated individual with the original one, and leaves the better one among them in the population.

The crossing process, shown in algorithm 3, takes a similar approach. It randomly pairs up the entire population, and for each pair, two crossing points in their genetic codes are randomly chosen, where their genes are severed and reconcatenated. As this procedure may cause redundancy where the same slot number appears twice in one individual, special care need to be taken. Lastly, as is in mutating, the better ones among the parents and children are left in the population.

Algorithm 3: DJS_cross

Input: ga
Output: ga
$ga' = ga$;
for $i = 1$ *to* $sample/2$ **do**
  Randomly choose two crossing points x_1, x_2 in $[2, slots-1]$ with $x_1 < x_2$;
  Randomly choose two individuals i_1, i_2 that haven't mated;
  Exchange the genetic code segments $[1, x_1]$ and $[x_2, slots]$ between i_1, i_2;
  Exchange any redundant element in i_1, i_2
end
for $i = 1$ *to* $sample$ **do**
  $ga(i) =$ DJS_time$(ga(i))$;
  if $ga'(i, slots+1) < ga(i, slots+1)$ *or* $ga(i, slots+1) = 0$ **then**
    $ga(i) = ga'(i)$;
  end
end

Lastly, the selecting algorithm (4) replaces the worst individuals with the best ones in each generation. All the individuals have a chance of survival to allow for genetic variety, the base probability for which is ps, but is minorly tuned according to the fitness of each individual: the longer the average finish time of an individual, the less likely that it will survive.

Algorithm 4: DJS_select

Input: $ga, select, ps$
Output: ga
$n = select$;
Find n individuals $a_1, \cdots, a_n$ with lowest avg_finish_time in the population;
Find n individuals $b_1, \cdots, b_n$ with highest avg_finish_time in the population;
for $i = 1$ *to* n **do**
  $p =$ rand()$*\frac{ga(a_i,slots+1)+ga(b_i,slots+1)}{2*ga(a_i,slots+1)}$;
  if $p \geq ps$ **then**
    $ga(b_i) = ga(a_i)$;
  end
end

4.3 Fitness

The last issue to deal with in our algorithm is the fitness function DJS_time, that calculates the average job finishing time of each sample. It first maps the permutation of slots to an allocation of jobs, then allocates a path for each data demand, and lastly calls averagetime() to calculate the average finish time with both job allocation and path allocation fixed.

Recall here that m stands for the number of jobs, while $D, \mathbf{d}, E, \mathbf{c}$ are the set of all data demands and their sizes, communication links and corresponding bandwidths respectively, and $\mathbf{t}$ is the execution time of each job.

Algorithm 5: DJS_time

Input: ga_r which is one row of ga
Output: ga_r

Number the processors in each data center concatenatively from 1 to $slots$;
for $i = 1$ *to* $slots$ **do**
  if $ga_r(i) \leq m$ **then**
    Allocate job $ga_r(i)$ to slot i;
  end
end
$P \leftarrow \emptyset$;
foreach $d \in D$ **do**
  Use BFS to find a path p from the data source to the site where the job is allocated;
  If there is no available path, **return** 0;
  $P = P \cup \{p\}$;
end
$ga_r(slots + 1) = \text{averagetime}(P)$;

Admittedly, algorithm 5 does not always yield the optimal average finish time given a fixed allocation of jobs, as the paths for data demands are allocated with a simple breadth-first-search. However, that is exactly the point of genetic algorithm: to use the simple, elegant imitation of natural evolution to complement the relative shortness of each individual, to use the uniformed approach that can be applied to almost all NP-complete problems to avoid the use of abstruse, sophisticated analytical techniques.

Algorithm 6: averagetime

Input: P
Output: $avgtime$

$finished = 0$;
$ft = \text{zeros}(m,1)$;
$totaltime = 0$;
while $finished < m$ **do**
  $path\_per\_edge = \text{zeros}(|E|, 1)$;
  $bandwidth = \text{zeros}(|P|, 1)$;
  foreach $p \in P$ **do**
    foreach $e \in p$ **do**
      $path\_per\_edge(e) + +$;
    end
  end
  foreach $p \in P$ **do**
    $bandwidth(p) = \min_{e \in p}\{c_e / path\_per\_edge(e)\}$;
  end
  for $i = 1$ *to* m **do**
    $ft(i) = \max_{d_{ij}}\{d_{ij} / bandwidth(p_d)\} + t(i)$;
  end
  Find job j with minimal ft;
  $totaltime = totaltime + ft(j)$;
  $finished + +$;
  Remove paths that ends at job j from P;
end
return $totaltime/m$;

And with both the job allocation and path allocation fixed, all that averagetime() needs to do is to guarantee the max-min fairness, which is basically making sure that all the paths passing the same edge share the same portion of its bandwidth.

5 Algorithm Performance and Analysis

5.1 Complexity

It is a well-known fact that the strengths of genetic algorithm lies in its simplicity and applicability, but not time efficiency. In fact, probabilistic method is called for when analyzing its time complexity, as the time it takes to execute is closely related to the probabilities of mutation and crossing, which counts for the most part of the time used.

Complexities of algorithm 5 and 6 are relatively easy to obtain. In DJS_time, mapping the permutation of slots to allocation of jobs takes $O(slots)$ time, while BFS, which takes $O(|E|)$ time, is called for $|D|$ times. In practice, we may consider $|E||D|$ to be much larger than $slots$, but since the links between data centers aren't that sparse, a breadth-first-search seldom takes $\Theta(|E|)$ time.

As for averagetime(), the main part of this function is a while loop, executed for m times. The two longer inner loops are executed for $|D|$ times, but the specific time required is closely related to how the source data is stored (for example, matrix or list), and how the software deals with matrices and vectors (like what level and how much of parallelism is implemented). However, as we are dealing with a practical problem, a practical method can be used to determine the bottleneck of the algorithm - the timing utility of MATLAB. The result of one demonstrative run is shown in figure 1.

函数名称	调用次数	总时间	自用时间*	总时间图 (深色条带 = 自用时间)
dis_job_sched	1	113.196 s	0.030 s	
DIS_time	11865	111.563 s	9.340 s	
averagetime	11865	97.774 s	97.774 s	
DIS_cross	48	90.222 s	0.726 s	
DIS_mutate	60	19.951 s	0.049 s	
BFS	1736859	4.449 s	4.449 s	

Fig. 1. time analysis

From the figure we can see that almost all the overhead comes from averagetime(), with DJS_time and BFS being the second and third time-consumer respectively. Further trials show that the time used by the later two is not asymptotically smaller that that of averagetime(), so suffice it to conclude that the time complexity of DJS_time is $O(\max\{m|D|, |E||D|\})$. (Note: in the actual program used to generate this analysis, some other data processing is done in DJS_time as well, resulting in its increased selftime.)

As for the rest part of the genetic algorithm, it is obvious (and experimentally proven) that the total time used is linearly proportional to the number of generations G. And from probabilistic method, in each generation the times that DJS_time is called is proportional to population size $sample$, and the sum of mutating probability pm and crossing probability pc. Figure 1 confirms this as well, as DJS_time is called for each mutated or cross-generated individual.

Thus, the overall time complexity of this genetic based algorithm for distributed job scheduling is roughly $O\left(G \cdot sample \cdot (pc + pm) \cdot \max\{m|D|, |E||D|\}\right)$, which isn't exactly polynomial since $G, sample, pc$ and pm are all numerical inputs. But we didn't expect it to be either, as we have proven in section 3 that this problem belongs to NP-hard class.

However, the inputs that break the polynomiality of the algorithm do not come innately with the scheduling problem, but are parameters set for the genetic algorithm, in a data-oblivious manner. And they do not need to increase with the problem size. With pc, ps and pm properly tuned, a population of several hundreds in size and several hundreds of generations should suffice for any input scale.

Lastly, since we are dealing with practical distributed systems in the first place, it should be noted that genetic algorithm has a natural affinity to data-level and thread-level parallelism. After mutating or crossing in each generation, the process of calculating each individuals' average finish time is mutually independent, and could be perfectly executed in a distributed way itself, significantly enhancing the time efficiency.

5.2 Performance

As is mentioned earlier, genetic algorithm does not guarantee an optimal solution, and the genetic parameters given in the previous text are chosen as a tradeoff between time efficiency and performance. The boundary is not quite clear, and the specification depends on the application backgrounds and demands. In general, increasing G and *sample* leads to a higher probability of better solutions, but increases time and space overhead as well. Increasing pm and ps, on the other hand, slows down the speed of convergence, but results in a better chance of jumping out of local optimality to the global one, while increasing pc and decreasing ps speeds up the process and can quickly leads the program to convergence.

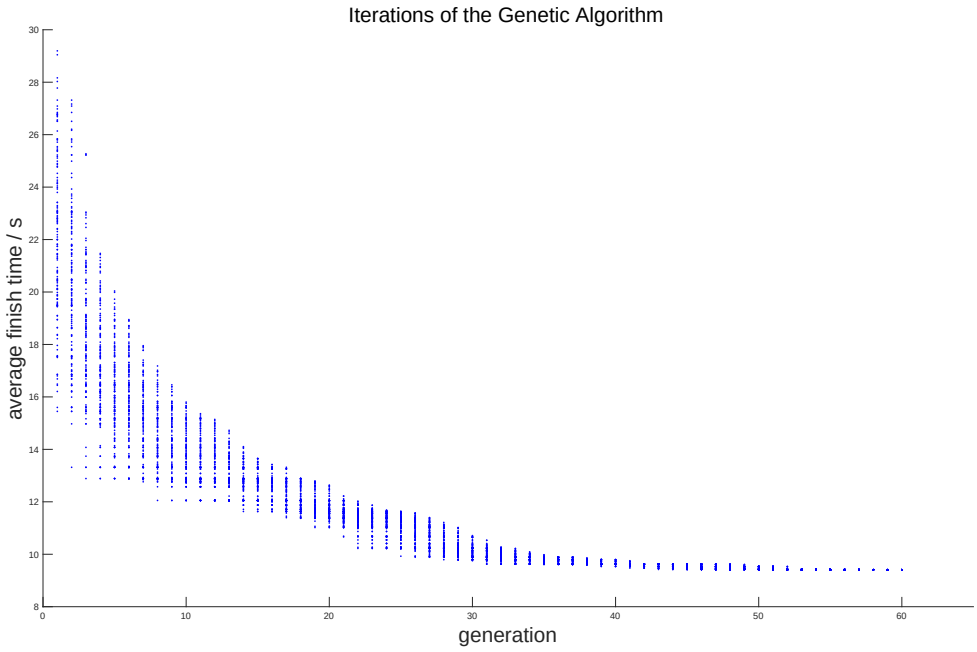

Fig. 2. evolution over generations

Figure 2 shows the process of convergence in an exemplary run with $G = 60$. This run yielded a result of 9.30s, the allocation for which is shown in figure 3, where each directed edge is a link in a path allocated to a data demand, with its width proportional to the demand's size. And the edge label indicates to which job the path leads.

This allocation is determinedly not optimal, as a bit of modification that sacrifices time efficiency in favor of performance (i.e. increasing the number of generations and the probability of mutation) yielded a better result of 8.79s. Nonetheless, it's close enough for pragmatic usage.

5.3 Testing

As a simulation to test our algorithm, we will also use randomly generated data. We assume that there are twenty data centers distributed around the world, in the countries with highest estimated nominal GDP in 2021 [8]. And there are one hundred computational slots in total, distributed among the countries proportionally to their GDPs. The links between these data centers are randomly generated with a rough probability of 1/3, with bandwidths ranging from 10 MB/s to 800 MB/s, and the inner-site bandwidth is 3000MB/s.

There are eighty jobs waiting to be processed, the number of data demands of each ranging from 1 to 5 randomly, of which 0-3 may be precedence constraints. Their execution time is also generated randomly from 0.5s to 5.5s.

Note that the jobs in this example are not grouped into clusters, and thus resulting in a far more complicated dependency relationship between each other than the previous example.

We processed this example using our algorithm, with a population of 150 and $G = 100$. The evolution and selection process of the population is shown in figure 4. It can be seen that a randomly generated

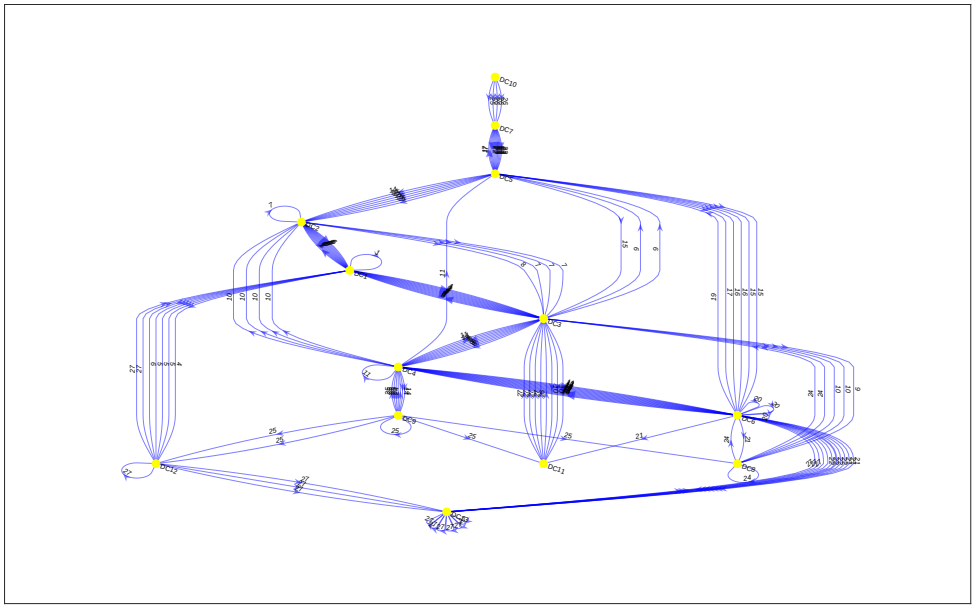

Fig. 3. the job and path allocation

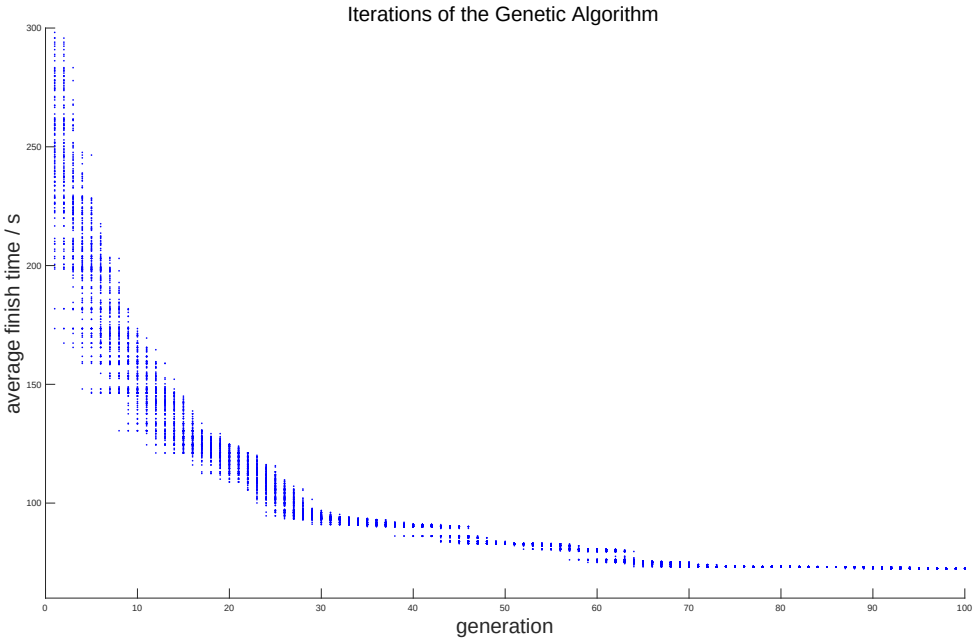

Fig. 4. evolution over generations

allocation has an average finish time between 200s an 300s with high probability, while our genetic algorithm yielded a solution with 72.4s in the end, the allocation of which is shown in figure 5. The whole program finished executing in several minutes, and without any other algorithm to compare with, we consider this a satisfactory performance.

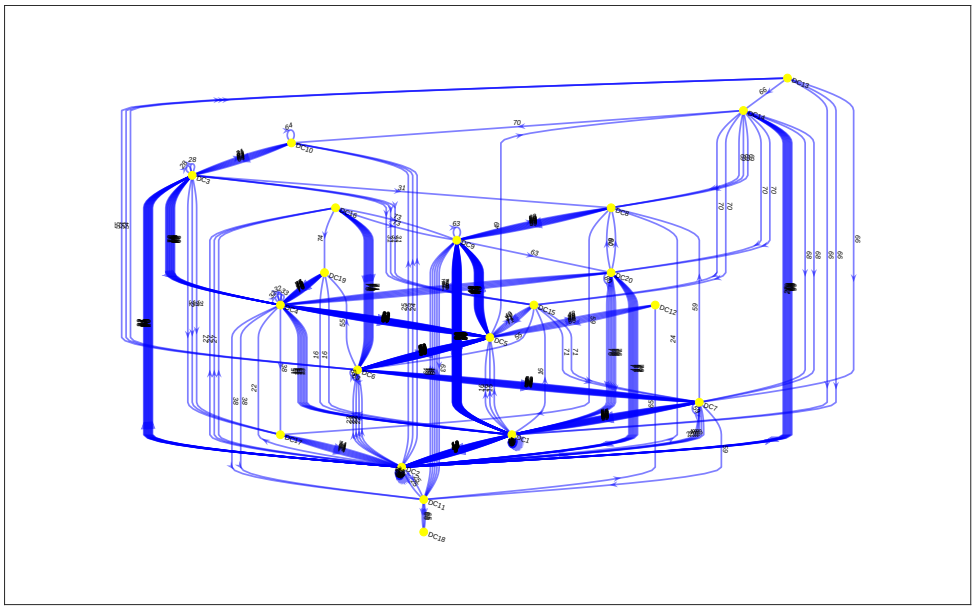

Fig. 5. the allocation of jobs and paths

Acknowledgements

Special thanks to Professor Xiaofeng Gao and Lei Wang from Department of Computer Science and Engineering, Shanghai Jiao Tong University, who guided the authors through the Algorithm and Complexity courses of this semester.

And special thanks to Professor Xiaodong Wu from School of Mechanical Engineering, Shanghai Jiao Tong University, who led the first author into the world of genetic algorithm during freshman Introduction to Engineering.

Given the limited time of the authors, this paper, along with the algorithm proposed in it, is admittedly far from perfect, especially concerning the behavior of the genetic algorithm near convergence, where there remains much to be optimized. Still, the authors gained much during the course of this project, and hold an appreciative attitude towards all those who participated or contributed to this problem. We look forward to any suggestions to the paper or improvements in the algorithm.

References

1. Bo Li: Scheduling Jobs across Geo-Distributed Data Centers. Department of Computer Science and Engineering, the Hong Kong University of Science and Technology
2. Chien-Chun Hung, Leana Golubchik, Minlan Yu: Scheduling Jobs Across Geo-distributed Datacenters. SoCC '15: Proceedings of the Sixth ACM Symposium on Cloud Computing, 111–124 (2015)
3. Thomas A. Roemer: A note on the complexity of the concurrent open shop problem. Springer Science + Business Media, LLC (2006)
4. Abraham Silberschatz, Greg Gagne, Peter B. Galvin: Operating System Concepts. 8th edn. Wiley, USA (2011)
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms. 3rd edn. MIT Press, USA (2009)
6. Matousek J., Nesetril J.: An Invitation to Discrete Mathematics. 2nd edn. Oxford, New York (2008)
7. Michael R. Garey, David S. Johnson: Computers and Intractability. W. H. Freeman (1979)
8. Wikipedia page, https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal).