Dynamic Hierarchical Structure Optimisation for Cloud Computing Job Scheduling

Peter Lane[1 [0000−0002−8554−2217]], Na Helian[1 [0000−0001−6687−0306]], Muhammad Haad Bodla[1], Minghua Zheng[1 [0000−0002−1909−8293]], and Paul Moggridge[1 [0000−0003−1004−1298]]

Department of Computer Science, University of Hertfordshire, Hatfield, UK
p.c.lane@herts.ac.uk, n.helian@herts.ac.uk, haadbodla@gmail.com, m.zheng2@herts.ac.uk, p.m.moggridge@herts.ac.uk

Abstract. The performance of cloud computing depends in part on job-scheduling algorithms, but also on the connection structure. Previous work on this structure has mostly looked at fixed and static connections. However, we argue that such static structures cannot be optimal in all situations. We introduce a dynamic hierarchical connection system of sub-schedulers between the scheduler and servers, and use artificial intelligence search algorithms to optimise this structure. Due to its dynamic and flexible nature, this design enables the system to adaptively accommodate heterogeneous jobs and resources to make the most use of resources. Experimental results compare genetic algorithms and simulating annealing for optimising the structure, and demonstrate that a dynamic hierarchical structure can significantly reduce the total makespan (max processing time for given jobs) of the heterogeneous tasks allocated to heterogeneous resources, compared with a one-layer structure. This reduction is particularly pronounced when resources are scarce.

Keywords: cloud computing · dynamic hierarchical job scheduling structure · genetic algorithms · optimisation

1 Introduction

Cloud computing has enabled users to quickly access information technology infrastructure, platform, and software from a pool of heterogeneous resources via the Internet. The requirements of cloud computing, from a user's perspective, include minimising response time to user's requests and reducing cost. As a relatively new technology, cloud computing has several open challenges, including service performance variation, quality of service, and energy consumption. Service performance variation is subject to divergence in task load and diversity of resources [18]. To mitigate the effect of variations in load on servers, effective load balancing requires efficient job scheduling and load migration techniques. This paper focuses on reducing makespan, the time to complete given jobs.

Scheduling jobs to minimise makespan and/or response time is an NP-complete problem [15]. Recent work considers a hierarchical structure in which jobs are

scheduled through sub-schedulers to servers. However, a fixed structure will not be optimal over time for all potential tasks. In this paper we introduce a dynamic solution to optimise the connections within such a job-scheduling structure in order to reduce the overall makespan, relying on an heuristic-search algorithm to find the optimal connection pattern.

A substantial amount of research has studied load balancing using two techniques [1, 14]. First, a load migrator (load balancer) is used to migrate workloads across different computing resources to improve the performance and reliability of cloud services by balancing the workload on different servers. In contrast, a job scheduler is used to assign tasks to servers proactively in order to balance workloads. Detailed explanations of load migration and job scheduling are given in the following two paragraphs.

Load migration algorithms define mechanisms for distributing incoming tasks across a cluster of servers. For instance, if a server is overloaded, then a part of the load on that server can be migrated to another server which has spare capacity. To ensure the most efficient and shortest execution time of tasks, workload should be balanced among all servers. Through load migration, the cloud system can optimise resource usage, increase throughput, reduce response time, and avoid being overloaded on some servers. Many load migration algorithms have been proposed. The literature [14] assigns various categories to those algorithms, including static or dynamic, centralised or decentralised, cooperative or non-cooperative, and intelligent or non-intelligent. Nevertheless, load migration might be very expensive in time cost due to an intrinsic problem. It has a three-stage process of information collection, decision making, and data migration. These often lead to an increased response time of task requests.

Job scheduling has the same aim as load migration but uses a different approach to reduce response time. A job scheduler takes a job which can consist of various tasks and assigns the tasks to servers. Job scheduling is done in a forward-looking way to reduce the time cost and implementation complexity of load balancing, and based on the estimated time of task execution. The main approach to job scheduling is to algorithmically dictate the order of task execution. There are a plethora of algorithms for job scheduling, including global level or local level, static or dynamic, meta-task (batch) or individual (online) [1]. Reducing service response time depends not only on the job scheduling algorithm, but also on the job scheduling architecture. In most research, one-layer scheduling is a commonly used structure for job scheduling [1], comprising job generation, job scheduling by scheduler, and task execution by servers. Some literature [4, 16, 17] indicates that hierarchical job scheduling structures, which do not dispatch tasks to servers directly, but via middle layers, could improve service performance. All these hierarchical job-scheduling structures are fixed/static, even though it is logical to think that better performance might be achieved using a more dynamic system. Some work has considered dynamic migration of tasks, but not dynamically scheduling tasks. No prior research has been conducted on dynamically altering the connections within a hierarchical job-scheduling structure based on the current status of jobs and servers.

The objective of this research is to design a dynamic hierarchical structure for job scheduling. The proposed design constructs a hierarchical structure, by adding a sub-scheduler layer between scheduler and servers to form a two-layer scheduling structure, and dynamically sets the connections to optimise efficiency. For a data center with a big cluster of servers, managing those servers in a hierarchical structure could improve resource use. In order to adjust the multi-layer structure to task changes, dynamic connections are implemented between the sub-scheduling layer and servers. To optimise the connections dynamically, artificial intelligent (AI) algorithms are used to address this NP-complete problem with the purpose of reducing makespan: here, we compare genetic algorithms and simulated annealing. Our design aims to mitigate the effects of performance variation for heterogeneous tasks and resources. Moreover, this design potentially accommodates other service requirements in a cloud computing environment, due to its dynamic and flexible nature.

2 Related Work

Earlier work on hierarchical structure design includes job scheduling and load migration for load balancing under a cloud or grid environment [4, 8, 11, 16, 17, 19]. Two pieces of work have been conducted on how dynamic design could balance workload and adaptively make use of available resources to achieve a better response time for user requests [3, 11]. In this section, we review some literature addressing two aspects of load balancing, through scheduling or load migration, with either a hierarchical structure or a dynamic design. First, we look at static hierarchical structures for job scheduling. Then we consider various dynamic systems which have been used in related applications, load migration, and similar areas of server capacity. As will be seen, none of these dynamic approaches covers the communication structure between the scheduler and servers.

A few studies have examined static hierarchical structure design for job scheduling [4, 9, 16, 17]. The three-level hierarchical structure used in [17] is designed to improve scheduling performance. Their results show that the proposed method can achieve better execution efficiency and maintain load balancing in the system. However, because the structure of the whole system is static, it may not be flexible enough to achieve the best task assignment when the tasks have a range of sizes. In [9], a similar static hierarchical structure design to [17] for job scheduling is presented. In this work, Jobs are assigned to Schedulers which then allocate the tasks to Workers. Although certain task allocation rules are applied between Schedulers and Workers in order to achieve low latency (similar to makespan), they cannot guarantee (approximate) optimal task assignment due to the nature of a static structure design. A three-layer structure to manage resources is also proposed in [4]. In this design, a heuristic algorithm is used to schedule tasks in the hierarchical structure. In the same way as [17], this structure is static. The authors in [16] designed a hierarchical structure consisting of a global placement manager and local placement managers. The global placement manager identifies local clusters of servers to be used based on workload

variability, and a local placement manager identifies the servers in the local pool based on the resource profile of the new workload and existing workload. Again, as with [17] and [4], the hierarchical structure is static with no optimisation.

A hierarchical management system designed for servers in a cloud is shown in [8]. Two types of servers are in this system: management and execution. A management server can dynamically change the structure. It solves the overutilisation problem by splitting a server node into two server nodes (the original server becomes one child and another child server is added). In contrast, it solves the underutilisation problem by removing a server node and distributing its children among its peers. The experimental results indicate that this hierarchical management system can improve response time and increase scalability. But this structure is on server side management, and the job scheduling structure is not explored. A dynamic hierarchical load balancing method that overcomes scalability challenges by creating multiple levels of load balancing domains is designed by [19]. This method considerably reduces the running time of the load balancing algorithm. This hierarchical structure is on servers and static. Similar to [8], the job scheduling structure is not investigated. Swarm optimisation algorithm is applied in [2] to migrate tasks across servers to achieve load balancing dynamically. Again, this work is focused on load migration, rather than job scheduling.

Some work has tried a dynamic approach to make a system adaptive to load changes. The authors in [13] proposed a set of heuristic scheduling algorithms, which could adapt to dynamical network states and changing traffic requirements, to maximise the network throughput by dynamically balancing data flow. Dynamic scheduling on the networking aspects of a sensor-cloud was proposed in [5] for optimising quality of service. Although their research proved that dynamic approaches could improve system performance, they did not consider load balancing on servers, which is the focus of our research in this paper. A dynamic heuristic algorithm for scheduling scientific workflow exhibits promising results in [12] but addresses a workflow scheduling problem. A new algorithm (Sigmoid) was proposed in [10] in order to dynamically load balance heterogeneous devices (servers), but the focus of this research is about how to split jobs to tasks which is different from ours.

The authors in [11] present a three-layer structure with the first layer being static, and the second and third dynamic. Their computational resources were dynamically available, and their algorithm could make best use of the resources. A decentralised approach which could migrate virtual machines by agent to achieve load balancing was devised in [3]. This paper developed a mathematical model for dynamic allocation of distributed resources in a data center. However, both of the aforementioned papers were focused on either solving the problem of load change or modelling the management of resource allocation. Neither of them investigated the way to schedule jobs.

All prior research has aimed at using static hierarchical structure or dynamic mechanisms for load migration, and to use static hierarchical structures or dynamic mechanisms for job scheduling. Those methods improve system per-

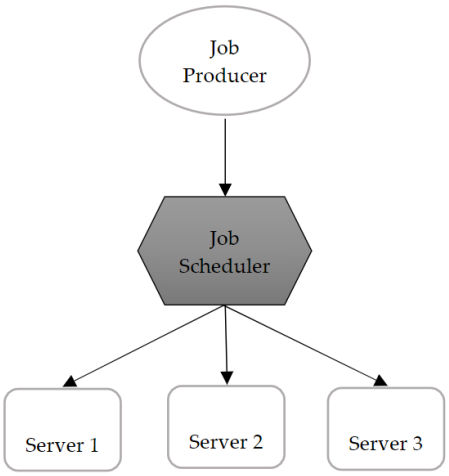

Fig. 1. One-layer structure

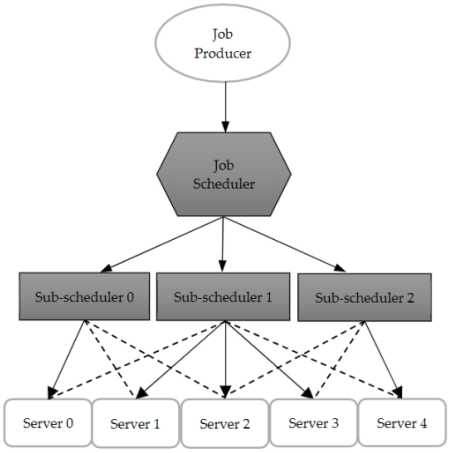

Fig. 2. Dynamic two-layer structure

formance in one way or another, but none of them is on designing dynamic hierarchical structure for job scheduling, which is the focus of the research presented in this paper.

3 Job Scheduling Structures

The scheduler accepts *jobs* from a Job Producer and allocates them (or their *tasks* to individual *Servers*. The scheduler can take many forms. A one-layer scheduler structure has a single scheduler and multiple servers. The scheduler takes jobs and then assigns tasks to servers for processing. As illustrated in Fig. 1, once jobs have been created by the Job Producer, they are passed on to the Job Scheduler. It is at this point where a scheduler algorithm, such as round robin [6], is deployed. Here, we take this well-used structure as a baseline against which to compare our dynamic two-layer structure.

In contrast to the one-layer scheduler structure, our two-layer hierarchical structure introduces a new sub-scheduler layer between the Job Scheduler and Servers. As illustrated in Fig. 2, the Job Scheduler receives jobs from the Job Producer and assigns each job's tasks onto the Sub-schedulers. Each sub-scheduler then further assigns tasks to Servers according to a scheduler algorithm. The connection pattern between the Sub-schedulers and Servers can be optimised to improve performance, such as measured by makespan. In addition, the connections are varied dynamically to adapt to any change of jobs. Fig. 2 uses solid lines to represent one possible connection pattern and dotted lines for another.

Our dynamic two-layer structure includes, as a special case, static hierarchical scheduling structures. For example, that introduced in [17] uses a Request Manager (Job Scheduler) to assign tasks to Service Managers (Sub-schedulers)

which divide those tasks into logically independent subtasks and subsequently load the subtasks to Server Nodes. Due to the static connection between the Service Manager layer and the Service Node layer, some assignments of subtasks may be suboptimal (e.g. if a small subtask is assigned to a Service Node with a large capacity). This structure from [17] is in the search space of our dynamic two-layer structure, which means our search algorithm can select it in cases when it is optimal or find an improved structure in cases where it is not.

4 Structure Optimisation

For our dynamic two-layer structure, achieving the best system performance requires the connection pattern between sub-schedulers and servers to be dynamically optimised to balance the loads across servers. Given a fixed number of sub-schedulers and a fixed number of servers, calculating all possible connections is an NP-complete problem. The optimal connection can be found by using a brute-force (BF) search algorithm, or through heuristic search algorithms, such as a genetic algorithm (GA) and simulated annealing (SA) algorithm. Makespan is the optimisation objective because, in general, makespan reflects the response time of user requests. The aim here is to demonstrate the effectiveness of dynamic optimisation of the connection patterns and so we do not, at this stage, consider network latency: future work will include network latency timings in the simulator and in the optimisation function.

In this optimisation context, the search space is not all the possible sets of connection patterns between sub-schedulers and servers because there are three constraints. First, each server can only be connected to one sub-scheduler. Second, each sub-scheduler must have at least one server. Third, the number of servers must be greater or equal to the number of sub-schedulers. For example, consider a system with three sub-schedulers, referred to as [0, 1, 2], and five servers, referred to as [0, 1, 2, 3, 4]. A connection pattern [2, 0, 1, 0, 1] indicates that Server 0 is attached to Sub-scheduler 2, Server 1 to Sub-scheduler 0, Server 2 to Sub-scheduler 1, Server 3 to Sub-scheduler 0, and Server 4 to Sub-scheduler 1. This is a valid connection pattern. But connection pattern [0, 0, 0, 0, 1] is not because Sub-scheduler 2 is not linked, and so it does not satisfy the connection constraints of our application.

The number of possible connections of the dynamic two-layer structure is:

$$S = P_n^m \times m^{(n-m)} \tag{1}$$

where S is number of possible connection structures, m is number of sub-schedulers and n is number of servers. P_n^m expresses randomly selecting m servers and then allocating one sub-scheduler to one server in a one to one relationship. The allocation order matters, so it is a permutation. For the remaining $(n-m)$ servers, each of them can be randomly allocated to any one of the sub-schedulers. Therefore the number of possible connection patterns for the remaining $(n-m)$ servers is $m^{(n-m)}$. Each allocation for the first m servers can be combined with

every single allocation for the remaining $(n-m)$ servers, hence we need to multiply P_n^m with $m^{(n-m)}$ to form the total number of connection patterns.

We consider three search algorithms for finding a 'good' connection structure.

4.1 Brute Force Search Algorithm

A brute-force, or exhaustive, search algorithm, explores a search space by testing every potential candidate solution. As this algorithm systematically enumerates all candidates to find the optimal candidate, it may take too long if the number of candidates is large. As per Equation (1), if m is 4 and n is 100, there are approximately 5.91×10^{65} candidate structures. Even with the most powerful computer in the world, it is impractical to examine all of them. Nevertheless, using a brute-force algorithm for smaller problems is a good way to compare the efficiency and accuracy of heuristic search algorithms of the same space.

Our brute-force search algorithm first sets the numbers of sub-schedulers, m and servers, n, before generating all possible structures connecting sub-schedulers and servers. Invalid structures are removed, based on the connection constraints of this application. After that, the algorithm checks each connection pattern to find the best candidate, the one with the smallest makespan.

4.2 Genetic Algorithm

Genetic algorithm (GA) uses a heuristic search, based on natural selection, to solve optimisation problems. A genetic algorithm iteratively modifies a population of individual solutions using operators such as selection, crossover, and mutation. For each iteration, GA selects from the better individual solutions in the current population (to be parents) and uses them to produce new solutions (children) in order to update the population. Over successive generations, the population evolves towards one containing even better solutions. The best solution is selected from the final population as the search result. GA is faster than the brute-force algorithm, especially when a search space is large, because it does not iterate through each candidate solution. However, GA may not precisely find the optimum, but an approximation.

In the experiments, a GA is used to optimise the connection pattern between sub-schedulers and servers. The GA works in a standard way, as follows:

1. **Initialisation**
   Four parameters are used. The population size, P, and the number of evolution cycles, e, control the GA process. The problem definition requires the number of sub-schedulers, m, and number of servers n. When producing each candidate, $m$schedulers are randomly allocated to the servers in a one to one relationship, then each of the remaining $(n-m)$ servers is randomly connected to one of the schedulers. This guarantees any generated pattern satisfies the connection constraints of this application.
2. **Fitness Function** The fitness of each individual connection pattern in a population is assessed using *makespan* as a fitness score.

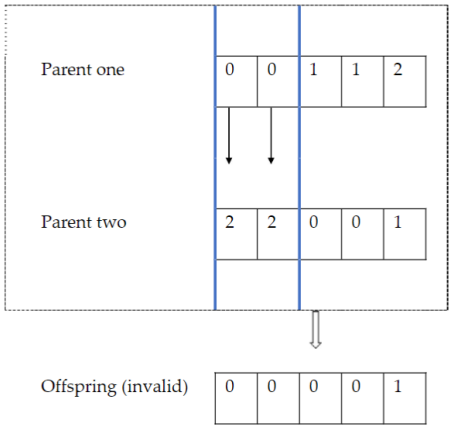

Fig. 3. Crossover (invalid)

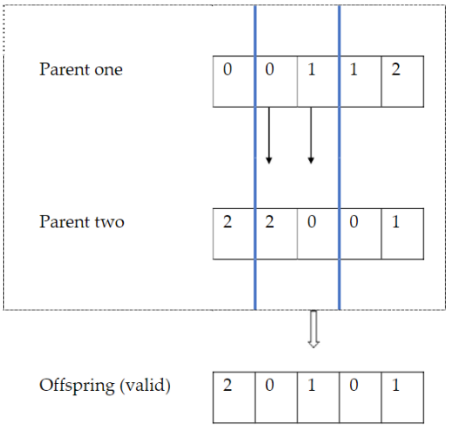

Fig. 4. Crossover (valid)

3. **Selection** The fittest individuals from the population are selected, so as to pass their best traits to the next generation, ultimately to improve the overall population fitness. The process of selecting an individual as a parent is to randomly select a fixed number (tournament size t) of individuals from the population, and then choose the best candidate on makespan from the selected pool. A pair of individuals (parents) is selected based on their fitness scores for generating an offspring.
4. **Crossover** Two selected parents are used to produce one offspring, respecting the connection constraints of the hierarchical scheduling structure. The crossover method first randomly selects a start and end element point from the array representing the parents' connection patterns. The end point must be greater than the start point. Then, the array elements from the start point to the end point of parent one are replaced by parent two's elements at the corresponding positions to produce an offspring. Due to the constraints of the connection patterns, an offspring might be invalid. If this is the case, a new pair of parents will be generated and the whole crossover process will start over until a valid offspring is produced.

   An example of this method is illustrated in Figs. 3 and 4. [0, 0, 1, 1, 2] and [2, 2, 0, 0, 1] are two single candidate connection patterns (parents). This structure consists of 3 sub-schedulers and 5 servers. In Fig. 3, the random start point is 0 and end point is 1. The elements between position 0 and position 1 of Parent one are then copied to Parent two at the corresponding elements to form an offspring [0, 0, 0, 0, 1]. This is an invalid offspring because it does not satisfy the connection constraints – Sub-scheduler 2 is not connected to any server. Hence, the crossover process is repeated until a valid offspring is generated. For example, with a start point 1 and an end point 2, the generated offspring [2, 0, 1, 0, 0] is valid, as illustrated in Fig. 4.

5. **Mutation** Some of the offspring are subject to a mutation on some of their genes (array element values): this process introduces some additional random variation into the population. A mutation probability parameter, p, determines the likelihood of mutating an offspring. As randomly changing the value of an element risks the new offspring becoming invalid, we mutate an offspring by randomly changing the position of each element.

Many factors can affect the performance and accuracy of a GA, including initialisation of the population of candidates, tournament size, crossover method, mutation method, mutation rate, termination condition, etc. The parameters of the GA used in this paper are given in Section 5.

4.3 Simulated Annealing Algorithm

Simulated annealing (SA) [7] is a widely applied optimisation algorithm, inspired by the process of heating and cooling material to reduce its defects. A temperature variable, T, is used to simulate the cooling process. When a simulated algorithm runs, T is initially set to a very high value. Later, the temperature is gradually reduced, like the cooling process of annealing, and candidate solutions are created and selected based in part on the current temperature. When the cooling process is finished, an approximately optimal candidate will have been found. SA is very simple to implement for complex problems compared with other evolutionary algorithms. Additionally, SA can avoid being stuck in local minima by (temporarily) accepting a worse variation as a new solution with a specified probability. In addition, SA can be very efficient because it does not iterate all candidates. However, in a similar way to GA, SA may not find the precise optimum, but a candidate which is very close to the optimum.

Here, we use a standard SA implementation:

1. The best pattern variable S_0 is assigned a random connection pattern.
2. C_0 is computed as the makespan of S_0, the current best solution makespan.
3. The new candidate S_1 is assigned a random connection pattern.
4. C_1 is computed as the makespan of S_1, the candidate makespan.
5. If C_1 is better than C_0, then accept the new candidate, setting $S_0 = S_1$.
6. Otherwise:
   (a) Compute acceptance probability p as $e^{\Delta E/T}$, where $\Delta E = C_1 - C_0$.
   (b) Generate a random number r
   (c) If $p > r$ then accept the new candidate, setting $S_0 = S_1$.
7. Reduce T by its cooling rate and repeat from step 2 as required.

When temperature is high, the chance of accepting a bad candidate is high; this means SA can explore broad regions of the search space and avoid becoming stuck in local minima. When the temperature is lower, the probability of accepting bad solutions reduces. The efficiency and accuracy of the simulated annealing algorithm depends on several factors, including candidate generator procedure, acceptance probability function, annealing schedule and initial temperature. For example, when the initial temperature value is set very high and

reduced slowly, the optimum has a greater chance to be found, but at the cost of speed. On the other hand, if the initial temperature is very low, the cooling process might become stuck at a local, sub-optimal minimum.

5 Simulation Experiments and Results

We empirically investigate under which scenarios a dynamic hierarchical structure outperforms a one-layer structure for job scheduling in a cloud computing environment, and demonstrate that heuristic-based search algorithms, such as GA or SA, can be effective in finding optimal structures.

We present results from a series of simulations designed to investigate whether or not our dynamic hierarchical structure could improve makespan over one-layer structure for cloud computing job scheduling, particularly in a heterogeneous environment. First, we compare three search algorithms for their ability to dynamically find the optimal connection pattern between sub-schedulers and servers. Second, we investigate the impacts of heterogeneous servers, non-uniformed tasks and scrambled job patterns on makespan: because the connection pattern is formed by optimising on these different continuations, the resulting job-scheduling structure will be optimised for these heterogeneous situations. This is done by looking at the impacts of server processing power (UPS) dispersion, task size (U) dispersion, and incoming waves of jobs, respectively.

5.1 Setup

A simulator is used to evaluate our job schedulers and compute their *makespan* values. Here, we use ScheduleSim[1], a lightweight, open-source simulator specifically designed for job scheduling in a cloud environment. We use the following definitions throughout this section:

Consumer(server) A consumer represents a virtual machine (VM) server executing tasks in cloud environment. The server simply reduces the units of a task according to its unit per step (UPS) rating. A "fast" server will have a high UPS and therefore be able to quickly deplete a task of its units.

Producer A producer represents the point at which jobs enter the Cloud; it also produces tasks and dispatches them to a scheduler.

Task A task is the simulated load on a system. Task is measured in unit U.

Scheduler A scheduler assigns tasks to sub-schedulers or servers.

Step ScheduleSim works using discrete time steps. Each entity (producer / scheduler / server) should step once for a time step to be complete.

Unit(U) Unit is a measure metric for the size of a task.

Each experiment creates tasks and servers using a random distribution. In these experiments, we use a normal rather than constant distribution in order

[1] P. Moggridge, ScheduleSim, 2019. https://bitbucket.org/paulmogs398/schedulesim

Genetic Algorithm		Simulated Annealing	
Initial population	50	Initial temperature	10,000
Mutation rate	0.015	Cooling rate	0.003
Tournament size	5		
Number of cycles	100		

Table 1. Parameter values used for GA and SA algorithms

Number of sub-schedulers	4	Task minimum size	20U
Number of servers	$\{4, 5, 6, 7, 8, 9\}$	Task maximum size	240U
Server pattern	Gaussian	Task size (μ, σ)	(200, 60) U
Server processing power (μ, σ)	(30, 0) UPS	Total Task Size	50,000U
Job pattern	Gaussian		

Table 2. Parameter values for the job scheduler

to obtain a heterogeneous set of tasks and servers. The continuous Gaussian distribution formula is used to create a distribution:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2}$$

where μ (mean) determines the position of the normal distribution, and σ^2 (variance) controls its width.

All experiments are measured using the metric of overall makespan, which is the latest finishing time of the simulation for given tasks.

$$\text{makespan} = \max_{j \in T} t_j \tag{3}$$

where, T is the set of all submitted tasks, and t_j is the time of task j to finish.

Each scheduler and sub-scheduler uses the round robin algorithm, which assigns tasks in circular order one by one. It is one of the most popular and simplest job scheduling algorithms, processing one task at a time, rather than many tasks at a time, so it is simple to implement and starvation-free. This makes the results easy to interpret.

The parameter values used for the GA and SA are presented in Table 1. (These values are given for the sake of reproducibility of results. In previous experiments a range of values was tried, and results were found to be robust to minor variations.) All results presented are the average of 20 runs to mitigate stochastic effects of the Gaussian distribution used to generate tasks and servers.

5.2 Experiment 1: Search algorithm comparison

This experiment compares three search algorithms (BF, GA and SA) on their ability to find good connection patterns. First, we want to know if the heuristic search algorithms can find solutions close to the optimum, and second get an idea of their efficiency. This experiment is on the hierarchical structure in Fig. 2.

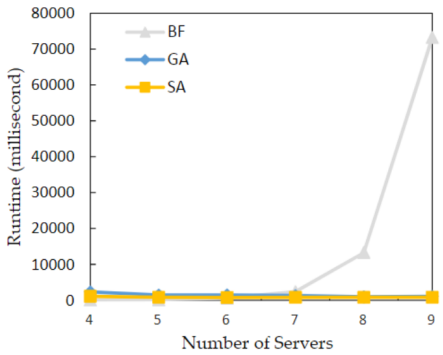

Fig. 5. Runtime for BF, GA and SA

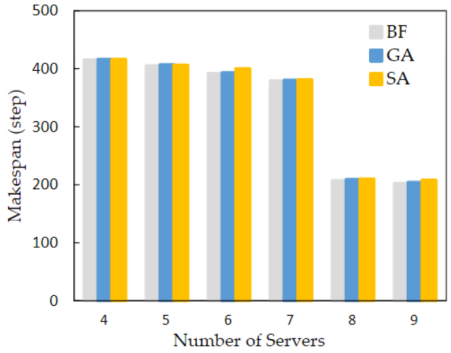

Fig. 6. Makespan for BF, GA and SA

Parameters defining the job scheduler are shown in Table 2. For this experiment, we used a Gaussian distribution for the job pattern, i.e. task size follows a Gaussian distribution and server processing power is uniform for all servers. Various server numbers are tested with the sub-scheduler number being fixed (as 4) for computational simplicity.

Results comparing the runtime and makespan, respectively, for the three search algorithms are shown in Figs. 5 and 6. In both graphs, the x-axis represents the number of servers and the y-axis the measured value. Runtime is the time taken for each algorithm to find the best connection pattern, and makespan is computed using the simulator.

As expected, the runtime for BF increases exponentially as the number of servers increases beyond 7, highlighting that it is not practical to use BF in real world applications. In contrast, the runtime for the GA or SA remains more or less the same, because these algorithms search a pre-set amount of the available search space, based on their parameters. For example, the GA always runs for 100 generations with a population size of 50.

Fig. 6 shows that both GA and SA can find the approximate minimal makespan, because both of them have nearly the same makespan values as the BF, which has performed an exhaustive search. As the runtime efficiency of the two AI algorithms is much better than the BF, particularly when the number of servers is large, it is reasonable to use one or other of these AI algorithms to find a dynamic connection pattern for our application.

In the real world, the number of servers is normally high, depending on the size of a data center, so we now perform some experiments on larger numbers of servers to investigate the relative efficiency of GA against SA. All the parameter values are the same as in Table 2, except the number of servers now takes values of $\{100, 200, 300, 400, 500\}$ and total task size is now 900,000U.

The results in Fig. 7 show that runtime increases slightly with server number; we attribute this to the increased need to delete invalid solutions during the search process. Moreover, it can be seen that the SA runtime is much shorter

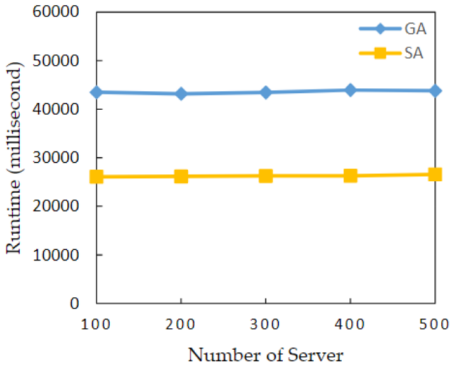

Fig. 7. Runtime on more servers

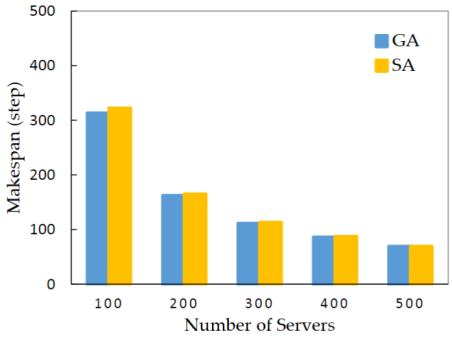

Fig. 8. Makespan on more servers

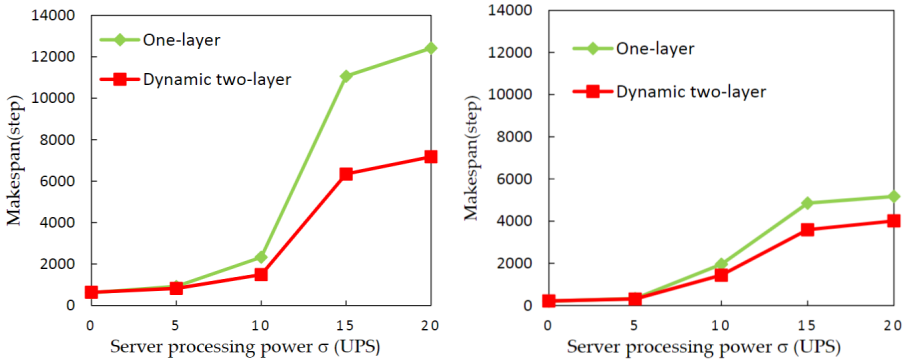

Fig. 9. Server dispersion impacts on makespan with 50 (left) and 150 (right) servers

(around 40%) than that of the GA. Furthermore, as shown in Fig. 8, makespan difference between the GA and the SA is small: the GA is slightly better initially, but the two are the same after the number of servers increases to 300.

5.3 Experiment 2: Server processing power dispersion impact

For exploring the effect of server processing power dispersion, we alter the following parameters from Table 2: Number of servers in $\{50, 150\}$, Server processing power μ is fixed at 30, $\sigma \in \{0, 5, 10, 15, 20\}$, Total task size 900,000U. The results (obtained from the GA) are shown in Fig. 9.

First, we note that the dynamic two-layer structure always outperforms the one-layer structure, although the makespan difference between these two structures is smaller with a larger number of servers; not shown here, but this holds over a larger range of server numbers, and is because optimisation becomes less important when there are more resources available.

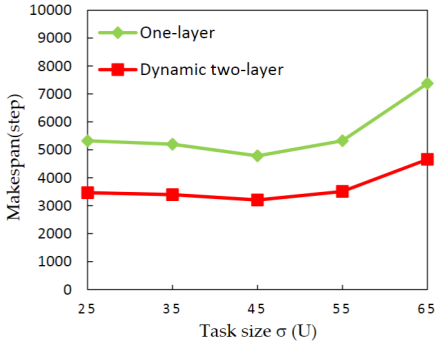

Fig. 10. Task size dispersion impact

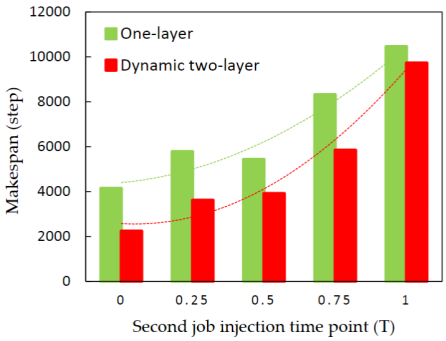

Fig. 11. Task complexity impact

Second, makespan lengthens with widening server processing power (larger σ), because it is harder to effectively assign tasks to heterogeneous servers than to uniform servers. Furthermore, the makespan improvement becomes bigger with σ widening, which implies that optimising the connection pattern is more important. The two results also indicate that the makespan improvement from a dynamic two-layer structure narrows with an increasing number of servers, which entails that optimisation is more necessary when resources are scarcer.

5.4 Experiment 3: Task size dispersion impact

For exploring the effect of task size dispersion, we alter the following parameters from Table 2: Number of servers is set to 100, Server processing power μ is fixed at 30, Task size $\sigma \in \{25, 35, 45, 55, 65\}$, Total task size 1,800,000U.

Fig. 10 demonstrates, again, that the makespan of the dynamic two-layer structure is smaller than that of the one-layer structure. In particular, makespan is relatively stable when σ is less than 45U, whilst the rate of makespan climbs after that. We could use an analogy to explain this. When there is a big size difference between a container and items, item occupation rate can be very high because small items fill gaps. On the other hand, when the item sizes are comparable to the container size, there would be more gaps left. Therefore, when the task size dispersion is small in a certain range, it is more likely that the servers are effectively occupied. However, when the dispersion is wider, there would be more tasks with the sizes comparable to server processing power, which might leave some servers underutilised. For this experiment, as compared to that in Table 2, we doubled total task size to generate more of the big tasks in order to show their impact (where the task size (U) is comparable to the servers processing speed (UPS)) on makespan.

5.5 Experiment 4: Job complexity impact

This experiment explores the impact of job complexity on makespan. Varying job complexity is achieved by injecting one job followed by another at various time

points in order to get different degrees of overlay on execution times between the two jobs. Both jobs have the same Gaussian distribution, hence individually they will execute in approximately the same amount of time T. Here, T is used as the unit of measuring injection delay of the second job. The first job is always injected at time point 0 and the second job follows with various delays. Therefore, the second job injection time of 0T implies that the two jobs are fully overlapped, 1T means no overlap and 0.5T entails 50% overlap.

For exploring the effect of job complexity, we alter the following parameters from Table 2: Number of servers is set to 50 and Server processing power μ is fixed at 30. Second job injection time point is in $\{0, 0.25, 0.5, 0.75, 1.0\}T$.

In Fig. 11, the makespan improvement decreases with the increase of delay on the second job injection. Smaller delay entails more job complexity, because the two jobs have a bigger overlap. In contrast, a bigger delay implies less overall job complexity because the two jobs have less overlap. Therefore, we argue that the more complex the jobs are, the bigger the improvement in makespan from using the dynamic two-layer structure. It is worth mentioning that makespan rises with the delay degree of second job injection. This is because execution completion time of the second job increases with the degree of its injection delay.

6 Conclusion

This paper presents a dynamic hierarchical structure, which introduces sub-schedulers between scheduler and servers, to improve job scheduling makespan in a cloud environment. This novel design dynamically changes the connection pattern between sub-schedulers and servers, using an heuristic search algorithm, such as a GA, to find the appropriate connection pattern for a given set of jobs.

In our experiments, we first show that both GA and SA achieve near optimal connection patterns. Second, simulation experiments demonstrate that a dynamic two-layer structure outperforms a one-layer structure on overall job makespan, particularly when assigning heterogeneous tasks to heterogeneous resources. This effect is particularly pronounced when resources are scarce. Due to its dynamic and flexible nature, this hierarchical design could potentially meet other system design requirements, including geographical resource distribution, as well as clustering resources and tasks for various purposes.

Future work will aim to improve the GA/SA design and parameter values, optimise sub-scheduler number and layer number, deploy performant scheduling algorithms (such as Max-min) on all schedulers and sub-schedulers, as well as taking into account additional objectives, such as quality of service, energy consumption and execution cost.

References

1. Arunarani, A., Manjula, D., Sugumaran, V.: Task scheduling techniques in cloud computing: A literature survey. Future Generation Computer Systems **91**, 407–415 (2019)

2. Balaji, K., Kiran, P.S., Kumar, M.S.: An energy efficient load balancing on cloud computing using adaptive cat swarm optimization. Materials Today: Proceedings (2021)
3. Benbrahim, S.E., Quintero, A., Bellaiche, M.: New distributed approach for an autonomous dynamic management of interdependent virtual machines. In: 2014 8th Asia Modelling Symposium. pp. 193–196. IEEE (2014)
4. Cao, J., Spooner, D.P., Jarvis, S.A., Nudd, G.R.: Grid load balancing using intelligent agents. Future generation computer systems **21**(1), 135–149 (2005)
5. Chatterjee, S., Misra, S., Khan, S.U.: Optimal data center scheduling for quality of service management in sensor-cloud. IEEE Transactions on Cloud Computing **7**(1), 89–101 (2019)
6. Dave, S., Maheta, P.: Utilizing round robin concept for load balancing algorithm at virtual machine level in cloud environment. International Journal of Computer Applications **94**(4) (2014)
7. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
8. Moens, H., Famaey, J., Latré, S., Dhoedt, B., De Turck, F.: Design and evaluation of a hierarchical application placement algorithm in large scale clouds. In: 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops. pp. 137–144. IEEE (2011)
9. Ousterhout, K., Wendell, P., Zaharia, M., Stoica, I.: Sparrow: distributed, low latency scheduling. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 69–84 (2013)
10. Pérez, B., Stafford, E., Bosque, J., Beivide, R.: Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems. Journal of Parallel and Distributed Computing **157**, 30–42 (2021)
11. Reddy, K.H.K., Roy, D.S.: A hierarchical load balancing algorithm for efficient job scheduling in a computational grid testbed. In: 2012 1st International Conference on Recent Advances in Information Technology (RAIT). pp. 363–368. IEEE (2012)
12. Sahni, J., Vidyarthi, D.P.: A cost-effective deadline-constrained dynamic scheduling algorithm for scientific workflows in a cloud environment. IEEE Transactions on Cloud Computing **6**(1), 2–18 (2018)
13. Tang, F., Yang, L.T., Tang, C., Li, J., Guo, M.: A dynamical and load-balanced flow scheduling approach for big data centers in clouds. IEEE Transactions on Cloud Computing **6**(4), 915–928 (2016)
14. Thakur, A., Goraya, M.S.: A taxonomic survey on load balancing in cloud. Journal of Network and Computer Applications **98**, 43–57 (2017)
15. Ullman, J.D.: NP-complete scheduling problems. Journal of Computer and System sciences **10**(3), 384–393 (1975)
16. Viswanathan, B., Verma, A., Dutta, S.: CloudMap: workload-aware placement in private heterogeneous clouds. In: Proceedings IEEE Network Operations and Management Symposium. pp. 9–16 (2012)
17. Wang, S.C., Yan, K.Q., Wang, S.S., Chen, C.W.: A three-phases scheduling in a hierarchical cloud computing network. In: 2011 Third International Conference on Communications and Mobile Computing. pp. 114–117. IEEE (2011)
18. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. Journal of internet services and applications **1**(1), 7–18 (2010)
19. Zheng, G., Bhatele, A., Meneses, E., Kale, L.V.: Periodic hierarchical load balancing for large supercomputers. The International Journal of High Performance Computing Applications **25**(4), 371–385 (2011)