The Journal of Supercomputing (2024) 80:15006–15037
https://doi.org/10.1007/s11227-024-06017-9

DiGTreeS: a distributed resilient framework for generalized tree search

Md Arshad Jamal[1] · Sriram Kailasam[2] · Bhumanyu Goyal[3] · Varun Singh[4]

Accepted: 20 February 2024 / Published online: 28 March 2024
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

Abstract
Exact combinatorial search algorithms have applications in several areas of computational algebra, AI, discrete optimization, etc. These problems are compute-intensive and have a highly irregular search tree. Most of the earlier efforts to parallelize these algorithms used a fixed degree of parallelism during runtime. We show that such an approach leads to poor resource utilization as the parallel run-time efficiency of an irregular search application varies over time. We propose DiGTreeS, a distributed resilient framework for generalized tree search that supports elastic scaling. It features an easy-to-use API for expressing combinatorial search and hides away the system concerns such as load balancing, fault tolerance, and elastic scaling. We evaluate the DiGTreeS framework for different scaling strategies and show its effectiveness on four representative problem instances: Traveling Salesman Problem, 0–1 Knapsack, N-queens, and Generic State Space Search Application.

Keywords Distributed computing · Elastic scaling · Irregular tree search · Fault tolerance

✉ Md Arshad Jamal
s20013@students.iitmandi.ac.in

Sriram Kailasam
sriramk@nitw.ac.in

Bhumanyu Goyal
bhumanyu.goyal08@gmail.com

Varun Singh
varunsinghs2021@gmail.com

[1] School of Computing and Electrical Engineering, Indian Institute of Technology, Mandi, India

[2] Department of Computer Science and Engineering, National Institute of Technology, Warangal, India

[3] Microsoft Corporation, Bengaluru, India

[4] Microsoft Corporation, Hyderabad, India

1 Introduction

Combinatorial search problems are found in several domains such as computational algebra, constraint programming [1]. These search problems can be classified into 3 categories [2]: (1) enumeration: where every solution that matches a certain property needs to be visited; (2) decision: to find out whether there exists a solution that satisfies a certain property; and (3) optimization: to find the best solution that optimizes a given objective function. These problems are highly compute-intensive and are characterized by a highly irregular search tree. Finding exact solutions for many such problem instances is NP-hard [3]. While approximation algorithms reduce the time required, they do not guarantee an optimal solution. An alternative is to speed up exact search by using parallelism where different parts of the search tree are explored in parallel. However, due to the highly skewed computation tree and the pruning heuristics that change the workload dynamically, parallelizing exact search is non-trivial.

YewPar [2] is a recent framework for parallel tree search that has implemented different search coordination techniques to improve the execution time. However, it uses a fixed degree of parallelism during runtime. As seen in Fig. 1, if we use a fixed number of workers (processing units) throughout the execution, the system efficiency decreases for different problem instances over time. Thus, it leads to poor resource utilization. In the cloud environment with a pay-per-use model, it translates into an increased execution cost. Elastic scaling can potentially reduce this cost; however, questions like when to scale, how much to scale, etc., need to be answered in the context of irregular tree processing. Another important requirement for tree search is the ability to recover from worker failures efficiently. As these tree searches are long-running jobs, without fault tolerance, the entire job will have to be rerun until successful completion.

Hitherto, frameworks like YewPar [2] have assumed a fixed set of workers to be available during the entire execution and do not offer support for elastic scaling. The fixed worker assumption may not be valid for long-running searches as computer systems can fail at any time. Thus, the desirable properties of a cloud-aware framework for parallel combinatorial search are (1) ease of expressing parallelism, (2) resilience to failures, and (3) support for elastic scaling. Most of the existing frameworks lack one of these. The most recent work, Equilibrium [4] supports (1) and (3) but not (2); HOPE [5] supports (1) and (2) but not (3). Existing related works are discussed in detail in Sect. 2.5.

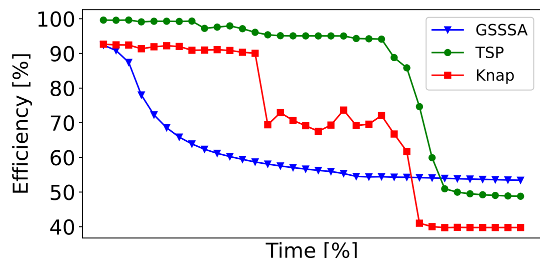


Fig. 1 Efficiency over time with the static number of workers (30) for different problem instances

This paper proposes DiGTreeS, a distributed resilient framework for generalized tree search. It has an event-driven architecture and is built using open-source resilient distributed services like Apache ZooKeeper, Apache Kafka, and HDFS (see Sect. 2.4). It provides an easy-to-use interface for specifying combinatorial tree searches while hiding away the system concerns. DiGTreeS implements an elasticity controller that monitors the system's elastic efficiency (see Definition 1) and dynamically adapts the number of workers to keep the system's elastic efficiency within a user-specified range using a reactive scaling strategy.[1] On detecting worker(s) failures, it recovers their state using checkpointing. We show the effectiveness of DiGTreeS on four representative problem instances: Traveling Salesman Problem and Knapsack (optimization), N-queens (enumeration), and GSSSA [7](enumeration). Compared to the state-of-the-art elasticity controllers Equilibrium [4] and Helpar [8], DiGTreeS has a higher benefit–cost ratio and it provides all three desirable properties of a cloud-aware framework for parallel combinatorial search.

The rest of the paper is organized as follows: Sect. 2 explains the relevant background and related work; Sect. 3 discusses the generalized API and walks through an example problem; Sect. 4 gives details of the system architecture and the elastic scaling algorithms implemented in DiGTreeS; Sect. 5 presents the experimental evaluation of the proposed framework; Sect. 6 discusses future directions; Sect. 7 discusses the conclusion.

2 Background and related work

2.1 Combinatorial tree search

Combinatorial search involves exploring a huge search space to find a solution. There are three major types of combinatorial search: (1) Optimization problems involve minimizing or maximizing the objective function within the search space. An example is the traveling salesman problem where the goal is to find a tour of all cities having the lowest cost. (2) Enumeration problems involve visiting each solution in the search space. An example is enumerating all the solutions of the N-queens problem where the goal is to place the n queens on an $n \times n$ chess board such that they do not attack each other. (3) Decision problems need to find whether there exists a solution that fulfills certain criteria; the search should stop as soon as any such solution is found. An example is the Boolean satisfiability problem [9] where the goal is to find assignments to variables in a Boolean formula that would make the formula true.

Combinatorial tree search algorithms deal with a system of discrete objects that can be configured into various states. They need to be arranged or selected in such a way as to achieve some cost function or to prove the existence of some combinatorial configuration. The naive way of finding these solutions is to start from the

[1] It is a technique where the elasticity controller reacts to the change in the system and makes decisions about scaling operations [6].

root node and keep adding children till we reach a leaf node. After processing the leaf, backtrack one step and add the next child. Keep repeating this process till all possible nodes have been explored. In combinatorial search problems, with brute force backtracking, the number of steps becomes exponential [2]. One technique for reducing the search space is branch and bound. Some parts of the search space can be dropped using a fast algorithm that calculates a bound on the best value possible in the current node's subtree. If this bound is worse than the best solution we have found so far, then we can prune the subtree. Such pruning operations cause the search tree to have a highly irregular structure.

2.2 Parallel combinatorial search

The different parts of the search tree can be explored in parallel by different workers. Since the search tree can be irregular, an initial distribution of the subtree among the workers may not ensure an even load. Hence, dynamic load balancing is required. Work stealing and work sharing paradigms have been explored for this [10]. In work stealing, an idle worker pulls tasks[2] from the task queue of a loaded worker, whereas in work sharing, the work is shared by the loaded worker with others. Among these, work stealing is found to be more communication efficient as communication is initiated only when a worker becomes idle [11].

For optimization problems, the workers share their best local solution discovered so far with the other workers. Using this, each worker can prune its subtrees whose lower bounds exceed the best solution. This helps faster pruning of the search space across all the workers. The final solution is the best-found global solution to the problem. For decision problems, the search is terminated when any worker finds an existential solution. However, to conclude that no solution exists, the entire search space needs to be explored. DiGTreeS uses a work-stealing-based approach for dynamic load balancing during runtime.

2.3 Elastic scaling in cloud environment

Cloud computing is attractive to high-performance computing (HPC) due to its pay-per-use policy and availability of huge compute resources. Elastic scaling is an important feature of cloud computing that allows it to scale processing units to respond to changing workload conditions [12]. The commonly used performance metrics are parallel execution time T_{par}, speedup (S), and parallel efficiency (E). These are computed for a problem instance P under the assumption of having a fixed number of processors (n) throughout the execution. Suppose T_{seq} is the execution time of a single-threaded sequential algorithm on problem P, then

[2] A task refers to an unexplored portion of the subtree.

$$S = \frac{T_{\text{seq}}(P)}{T_{\text{par}}(P)} \tag{1}$$

$$E = \frac{1}{n} \times S \tag{2}$$

Henceforth, we drop P from the above formulae to make them easy to read. With elastic scaling, the number of processors changes during the execution. So, the basic assumption of the number of processors being constant in the calculation of parallel efficiency gets invalidated. Moreover, to calculate parallel efficiency the entire execution needs to be finished. Hence, there is a need for a metric that can be obtained at run time and can be used to make decisions related to scaling.

Equilibrium [4] uses the concept of essential and non-essential computations. DiGTreeS borrows this concept from Equilibrium [4], but the underlying mechanism to compute the time spent in the essential computation is different. Equilibrium [4] relies on `ThreadMXBean`, the management interface for the thread system of the Java virtual machine (JVM), for measuring the threads'/workers' CPU time,[3] Which they consider as time spent in essential computation, whereas DiGTreeS takes care of the measurement of computation times by recording all the duration for which a worker is involved in the essential computation.

Computations performed by all the workers that are also performed in the case of sequential execution are essential computations. Time spent in communication, work transfer, worker(s) being idle, etc., are all non-essential computations. The efficiency of a particular worker is the ratio of time for which it is involved in essential computation to the total time elapsed. We call run-time parallel efficiency as elastic efficiency and denote it as E_{elastic}. We define E_{elastic} as follows.

Definition 1 *(Elastic Efficiency)* The elastic efficiency (E_{elastic}) is the arithmetic average of individual efficiencies of all the processing units (workers), where the efficiency of a processing unit is the ratio of time for which it is involved in essential computation to the total time elapsed.

Mathematically, at any given point of time t, E_{elastic} is calculated using Eq. 3, where E_i is efficiency of a worker i and n is the number of processing units at time t.

$$E_{\text{elastic}}(t) = \frac{1}{n} \times \sum_{i=1}^{n} E_i \tag{3}$$

DiGTreeS uses E_{elasctic} as the metric for making decisions related to scaling operations (see Sect. 4.3).

[3] ThreadMXBean returns the user-level CPU time for the current thread if CPU time measurement is enabled; −1 otherwise.

Let s_i and e_i denote when a worker got provisioned and de-provisioned, respectively. As the cloud uses a pay-per-use policy, if C_i denotes the cost per unit time of using worker i, then the total cost of elastic scaling (C_{es}) can be calculated as:

$$C_{\text{es}} = \sum_{i=1}^{n} C_i \times (e_i - s_i) \tag{4}$$

We use this cost metric and execution time to compare the performance of different scaling strategies in DiGTreeS.

2.4 Overview of frameworks used in DiGTreeS

Apache ZooKeeper[4] [13] is a distributed coordination service that is used to implement protocols like master election, notification on process failures, locking, etc. It provides a hierarchical namespace and allows storing small amounts of data in znodes. These can be read or written atomically. The 3 types of znodes are persistent: store data persistently unless explicitly deleted; ephemeral: retain only as long as the session is alive; sequential: an increasing sequence number assigned automatically based on the creation time. ZooKeeper supports event notifications by setting a watch on a particular znode that gets triggered when the znode is deleted, its data are changed, or its children are changed.

Apache Kafka[5] [14] is a reliable distributed high-throughput messaging system. It gives the abstraction of a topic to which messages can be posted by producers and the consumers subscribed to that topic can read those messages. DiGTreeS uses Kafka for reliable high-throughput work transfers.

Hadoop Distributed File System[6] [15] (HDFS) is a fault-tolerant distributed file system based on master/slave architecture. It provides high-throughput access and is designed for storing large datasets. DiGTreeS stores snapshots in HDFS which is used for fault tolerance.

2.5 Related work

Gupta et al. [16] identify communication overhead and synchronization requirements of irregular tree search as one of the main reasons for its poor performance on the cloud. They suggest over-decomposition of tasks and overlapping computation with communication as possible solutions. In DiGTreeS, communication and computation are overlapped, but there is no over-decomposition of tasks. The tasks are generated by the individual workers as part of their tree search and idle workers can request work from other busy workers. This reduces the task tracking overhead. Work queue+[17] uses the Work Queue framework for building parallel cloud-aware applications. This work supports elasticity and handles worker failures by reassigning failed tasks to others. However, this approach is master-heavy. Thereby, it can act as a single point of failure and a potential bottleneck to scalability. In contrast, the task list

[4] https://zookeeper.apache.org/.

[5] http://kafka.apache.org/.

[6] https://hadoop.apache.org/.

is maintained in a distributed manner in DiGTreeS; thus it is master-light and scales well. Kehrer and Blochinger [18], Haussmann et al. [7], Rosa Righi et al. [4] have proposed different designs for elasticity controllers. Rosa Righi et al. [18] present a reactive elasticity controller for iterative applications. Their scaling function is based on an exponential weighted averaging of the measured efficiencies over time. Strictly speaking, parallel tree search applications are not iterative in nature; however, their scaling function can still be used. Haussmann et al. [7] present a more sophisticated cost-based auto-scaling approach considering the low-cost opportunity instances in the cloud. Equilibrium [4] presents a runtime efficiency metric that distinguishes between essential and non-essential computations and adapts the number of instances to meet user-specified target efficiencies. DiGTreeS also uses the time spent in essential computations to measure the runtime efficiency, but the scaling mechanisms and efficiency computations are different (see Sect. 4.3 for more details).

The problems on parallel tree search and dynamic scaling have also been explored in cluster computing. Archibald et al. [2, 19], Poldner and Kuchen [20], Bungart and Fohry [21] present generic skeletons for expressing parallel tree search. Poldner et al. [20] and Bungart et al. [21] implement dynamic scaling using MPI and x10[7], respectively. However, the goal of dynamic scaling in cluster computing is to optimize resource utilization at the cluster level considering all the jobs that are currently executing, whereas, in the cloud environment, it is application-specific.

2.6 Problem instances

There exists a wide range of combinatorial problems to consider for benchmarking and testing the proposed framework. We consider three classic problem instances and one problem instance from the recent works in exact combinatorial searches for benchmarking and testing DiGTreeS.

(1) *Traveling Salesman Problem:* The Traveling Salesman Problem (TSP) is a well-known NP-hard problem [22]. Given a set of cities and the cost incurred in traveling between any two cities, a salesman has to visit each city exactly once and return to the starting city. The objective is to find such a path that incurs minimum cost. It is implemented as an optimization problem.

(2) 0–1 *Knapsack Problem:* Knapsack (Knap) Problem is also an optimization problem. Given a set of items, each with a weight and a value, we need to find the maximum total value of items that can be put inside a knapsack of a given capacity [23]. It is implemented as an optimization problem.

(3) *N-Queen Problem:* N-queens (NQ) are an enumeration problem. Given a chessboard of size N×N, find a way to place N queens on the board such that no two queens can attack each other [24]. It is implemented as an enumeration problem.

(4) *Generic State Space Search Application:* GSSSA is a benchmark application as proposed in [7]. It allows us to control the irregularity of search trees with a

[7] http://x10-lang.org/

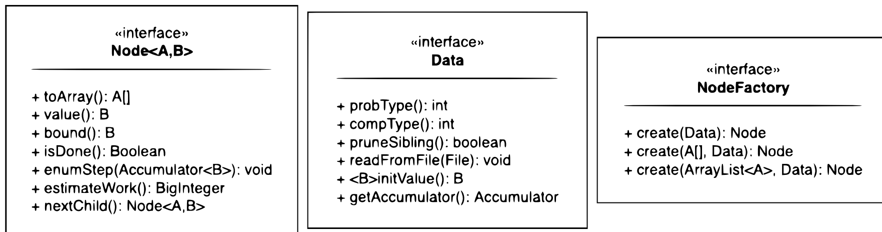


Fig. 2 Generalized API of DiGTreeS

small set of parameters while exhibiting relevant features of parallel tree search applications. GSSSA creates a binary search tree where each node has a workload value W representing a number of random SHA-1 hash calculations to be done as work. Each node can either act as a leave and do hash calculations or divide into two child nodes. The root node divides into two nodes creating two separate subtrees, one regular fraction with workload W_r and another irregular fraction with workload W_i. For further details on GSSSA refer [7]. It is implemented as an enumeration problem.

In Sect. 1, we discussed that combinatorial search problems can be classified into 3 categories, (1) enumeration, (2) decision, and (3) optimization. In the case of TSP, getting a path with minimum cost is an optimization problem. The corresponding decision version is whether there is any path possible that covers all the cities with cost = $\mathcal{K}$. Similarly, to get all the valid TSP paths with cost = $\mathcal{K}$ is an enumeration problem. Again this can be converted into a decision problem by terminating the search the moment a valid path with cost = $\mathcal{K}$ is found. Similarly, a combinatorial problem of any type can be converted into any other type. In the experimental evaluation section, we showed results for optimization and enumeration problems only.

3 Generalized application programming interface (API) for tree search

DiGTreeS provides a generalized interface for specifying tree search-based problems (see Fig. 2). There are 3 abstract classes that the user needs to extend for specifying a new problem. The abstract class `Node` defines the details of a specific node in the tree; `Data` are for the data that is common to all the tasks in the problem; `NodeFactory` is for creating an instance of the `Node` type. With the help of these interfaces, DiGTreeS covers the requirements for a wide range of problem types—optimization, enumeration, and decision. Java generics are used to support different data types.

In the `Node` interface, `A` refers to the serialization type used for work transfer, while `B` refers to the solution type. The `value` method returns the solution at that node if a solution exists (`isDone` is true); `estimateWork` method should return an approximate value of the total work in exploring the subtree below that node (see Sect. 3.1); `nextChild` should return the next non-visited child of the current node.

The `enumStep` method takes an object of type Accumulator that can be updated for enumeration problems. The `bound` method can be used for setting a best-case bound for the current exploration sub-tree. In the `Data` interface, an initial estimate of the solution can be specified using the `initValue` method; the `probType` and `compType` indicate the problem type, and the comparator type for optimization/decision problems; the `readFromFile` method specifies how to parse the input data. If the `pruneSibling` is enabled then the siblings of the child node are also discarded in case the child node does not satisfy the bound. This is useful for the Knapsack problem as the heuristic ordering ensures that an item with a better ratio of value/weight gets processed before the rest of its siblings. The possible children at any step are nodes s.t. the sum of their weights is less than the capacity. Some of the methods in the interfaces are specific to the type of problem ((1) optimization, (2) decision, and (3) enumeration) being implemented. `bound` method requires implementation only in case of an optimization problem. Implementation of the `value` method is supposed to return the solution for the current node for the problems of types (1) and (2). The method `isDone` is also associated with the problems of types (1) and (2). The method `enumStep` is required for the problems of type (3).

3.1 Work estimation and bounds calculation

The estimate of the remaining work at each worker helps the load balancer make better decisions [25] while evaluating the bounds helps in pruning the search space. As work estimation and bound evaluation are problem-dependent, the methods `estimateWork` and `bound` are to be implemented for a problem instance. These are part of the `Node` interface in the DiGTreeS API (see Fig. 2). We show an example of how these methods can be implemented for the Traveling Salesman Problem (TSP).

In Sect. 2.6, we discussed TSP. The brute force way to solve the optimization version of the TSP is to visit the cities using each possible permutation of cities and find the path with the minimum cost. The complexity of this approach is $\mathcal{O}(n!)$. The branch and bound optimization can be used to improve the run time of this problem. A bound on the path length can be calculated using the minimum spanning tree algorithm [26, 27], which can be used to trim the unnecessary branches' exploration. For the initial estimate of the solution, we used the nearest neighbor algorithm [28], and for the work estimate, Eq. 5 (i = No. of unvisited cities) provides a good approximation of the expected required work to solve the current task. Unpacking the task is 1 unit of work and then we need to solve i tasks each of size $W_{est}(i-1)$. The weight of the minimum spanning tree can be used to calculate the bound for TSP.

$$W_{\text{est}}(i) = \begin{cases} 0, & \text{if } i = 0 \\ W_{\text{est}}(i-1) * i + 1 & \text{otherwise.} \end{cases} \tag{5}$$

GSSSA [7] has been implemented as an enumeration problem, so it does not require the `bound` method to be implemented. Each node in the exploration tree contains the data related to regular workload (W_r) and irregular workload (W_i). Method `estimateWork` returns summation of W_r and W_i. Likewise, the `bound`

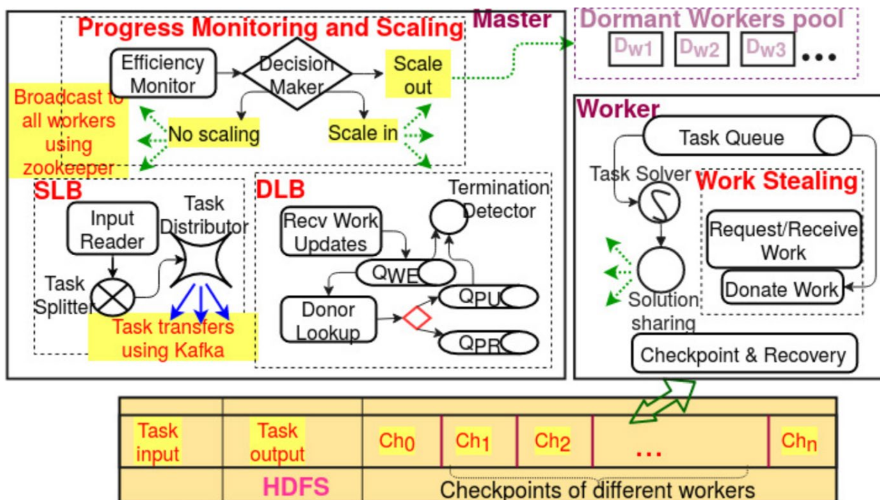


Fig. 3 System architecture of DiGTreeS

(if required) and `estimateWork` methods can be implemented for a problem instance. Similarly, if any new problem comes it needs to be implemented using the DiGTreeS API (see Fig. 2). For evaluating the proposed framework DiGTreeS, we have implemented four problem instances as mentioned in Sect. 2.6. All four implementations are problem-dependent and not framework-dependent. By no means do we claim that these implementations are better than the state of the art. These are merely some of the ways to implement problems using DiGTreeS APIs. Implementations are based on trivial branching and pruning functions and work estimations. It shows the generality of the approach but is far from the state of the art. There exist or may exist ways to get better work estimates or calculate a better branch-and-bound condition. These implementations are done to show that DiGTreeS APIs can be used for any combinatorial problem. For instance, in practice, TSP is often solved using linear programming and cutting plane techniques (branch and cut). For comparison with a commercial Integer Linear Programming (ILP) solver, we implemented TSP in CPLEX.[8] CPLEX performs better than DiGTreeS. But any ILP is just for mathematical optimization problems. It is not a generalized thing. For an input of 35 cities (see Sect. 5 for details on input values), our TSP implementation took around 700 s, whereas CPLEX took around 302 s.

4 DiGTreeS

DiGTreeS is implemented on top of Apache ZooKeeper, Apache Kafka, and Hadoop Distributed File System (HDFS). ZooKeeper supports several common patterns of coordination in distributed systems. In DiGTreeS, master election protocol,

[8] CPLEX is a commercial ILP solver by IBM.

monitoring events such as worker failures, changes in data structures, worker load estimates, and solution sharing are implemented using ZooKeeper. Kafka is used for performing reliable task transfers between donor and requester, and between master and worker. HDFS stores the worker snapshots reliably, which is used for recovery.

4.1 System architecture

DiGTreeS follows a master-worker architecture (see Fig. 3).

4.1.1 Components of master

The master has 3 main components that are responsible for (1) static load balancing (SLB), (2) dynamic load balancing (DLB), and (3) progress monitoring and scaling.

(1) *Static Load Balancing:* SLB is performed by the master (Lines 4–6, Algorithm 1). It starts with a queue that has only the root node (which needs to be specified in the problem's implementation). It pops the first element, converts it into its children using `nextChild()` operation specified in the problem's implementation (see Fig. 2), and pushes them back into the queue. If the size of the queue becomes larger than the number of workers, then it removes the elements from the front and assigns them to different workers. This process is repeated until the remaining task size becomes very small. The remaining work is given to the first worker. Once the SLB phase is completed, workers are notified through ZooKeeper to start exploration, and the elasticity controller subroutine is invoked (Line 7, Algorithm 1).

(2) *Dynamic Load Balancing:* The DLB component maintains a priority queue of work estimates for each worker to implement an efficient donor lookup scheme. It handles different types of requests (lines 8–27, Algorithm 1). When an idle worker requests work from the master, it returns the most loaded worker from the priority queue as a donor and decreases the donor's load by half. This prevents a single donor from being swamped by too many requesters. The requester then contacts the donor directly for work and after receiving work, it updates the master about its new workload. Until that update is received, the master stores them in a pending updates queue Q_{PU} (`pendUpd` in Algorithm 1). It may happen that the master's overall estimation of system load becomes zero although there is still work available. This can happen when the requester has not yet updated the master, and before that update, the other workers finished all their work. If a worker requests a donor at this time, then it is added to the pending requests queue Q_{PR} (`pendReq` in Algorithm 1). These pending requests are served when the master receives a nonzero work update from any worker. At the end of the computation, the master's overall estimation of the system load becomes zero. When this condition occurs, the termination detection component sends the termination signal to all the workers (Lines 30–32, Algorithm 1). The master-assisted DLB scheme in DiGTreeS not only reduces the latency of locating a suitable donor but also allows the master to remain lightweight as the

master only stores metadata and not the actual tasks. `DonorRequest()` and `WorkUpdate()` methods in Algorithm 1 help in DLB.

(3) *Progress Monitoring and Scaling:* This component computes $E_{elastic}$ using updates sent by the individual workers. It then determines the mode of operation: scale-out/scale-in/continue as is (normal). For scale-out, workers are provisioned by the master, whereas for scale-in the workers receive notifications to drop out. In Sect. 4.3 and Algorithm 3, the progress monitoring and scaling component is discussed in detail.

For a detailed discussion on the Master Algorithm 1, refer to our earlier work RD-FCA [25].

Algorithm 1 ALGORITHM FOR MASTER

```
Procedure MASTER(masterId)
 1: Initialize: aliveWorkers, workEstimate, masterState
 2: pendUpd ← [], pendReq ← []
 3: announce masterId as master
 4: if masterState=start_state then
 5:   performSLB(aliveWorkers)
 6:   update masterState ← SLB_Done in ZooKeeper
 7: Invoke Elasticity Controller subroutine // Algorithm 3
 8: upon DonorRequest(requesterId):
 9:   workEstimate.update(requesterId, 0)
10:   pendUpd.remove(requesterId)
11:   if isTerminated() then
12:       terminate(requesterId)
13:   else
14:       donor ← workEstimate.first() // Worker with max. work
15:       if donor.est > 0 then
16:         pendUpd.add(requesterId)
17:         assignDonor(donor.wId, rId)
18:       else
19:         pendReq.add(requesterId) // No donor available

20: upon WorkUpdate(requesterId, est):
21:   pendUpd.remove(requesterId)
22:   workEstmate.update(requesterId, est)
23:   if est > 0 then
24:       serve requests from pendReq
25:   else
26:       if isTerminated() then
27:           terminate(pendReq)

28: upon WorkerFailure(workerId):
29:   remove workerId from pendUpd, pendReq, workEstimate
      and aliveWorkers.
30: upon TerminationDetection():
31:   if workEstimate.first().est == 0 and pendUpd.isEmpty() and
        recoveredAll() then
32:         send a termination signal to all alive workers.
```

4.1.2 Components of worker

The worker has components for (1) work stealing, (2) solution exploration, (3) solution sharing, and (4) checkpointing and recovery.

(1) *Work Stealing:* The work-stealing component is responsible for requesting work from a donor, donating work to other requesters, receiving work from the master during SLB, and updating the current workload as well as the local efficiency to the master. Each worker uses the load-estimator function specified by the user and reports its load to the master only when its load reduces by half of its current value. This reduces the network traffic for work updates to log(*initial_load*) and works well in practice.

(2) *Solution Exploration:* See Algorithm 2. Each worker stores the tasks received during SLB into a task deque. Based on the problem type, an appropriate function is called for solution exploration. If the problem type is optimization (for example, minimizing objective function), then `ExploreOptMin` (Line 1) function is called. At each exploration step, a node from the front of the deque is popped and its children nodes are generated (Line 2). Next, the solution bound in the subtree of the child node is compared with the current best solution found globally (`globalBest`). For example, if the goal is to find the minimum value, then the lower bound of the solution in the subtree of the child is considered. If that bound is lower than the current `globalBest`, then only the child node is explored; otherwise, it is discarded. The bound function is specified by the user as part of the `Node` Interface. If the solution is reached at the child node and is lower than the current `globalBest` value, then the global best value is updated. Otherwise, if the `pruneSibling` flag is enabled, then the siblings of the current node are discarded (Lines 3–8). A similar function with comparisons reversed is written for maximization problems. The exploration method for a decision problem is similar to the optimization problem, with the only difference being when a suitable solution is found, a signal is sent to all the workers to terminate the search using ZooKeeper.

For enumeration problems (Line 9), an accumulator is used to collect information from each node. A new accumulator object is created for each worker and is passed to the `Node.enumStep` method. Inside this method's concrete implementation, the user can choose to update the value of the passed accumulator based on the problem. For example, for the N-queens problem, we use an Integer accumulator and increment it at each `enumStep` to count the total number of solutions.

Algorithm 2 Exploration methods

```
1: Function EXPLORE_OPT_MIN(Node n):
2:    for child in n.generateChildren() do
3:        if child.bound() < globalBest then
4:            if child.done() then
5:                updateBest(child)
6:            addToDeque(child)
7:        else if pruneSibling then
8:            break

9: Function EXPLORE_ENUM(Node n):
10:   for child in n.generateChildren() do
11:       child.enumStep(this.accumulator)
```

(3) *Solution Sharing:* This component is invoked by the solution explorer to read/write to the current best solution found globally (among all the workers). The sharing component does not send an update to the globally shared value for each update made by the solution explorer. Instead, it maintains a local copy and updates that. Periodically (in a few seconds granularity), it syncs with the global value. This is to reduce the network traffic. Solution sharing is implemented using ZooKeeper watches.

(4) *Checkpointing and Recovery:* DiGTreeS uses an asynchronous snapshot mechanism wherein each worker takes a snapshot of its task queue and writes it to HDFS atomically using techniques similar to shadow paging. Each worker's snapshot is stored in a different directory in HDFS. These snapshots are used during recovery. DiGTreeS guarantees at least once semantics. For failure detection and recovery, it sets up a circular queue consisting of alive workers in which each worker monitors its successor for failure using ZooKeeper. On detecting a worker failure, the monitoring worker reads the latest snapshot of the failed worker from HDFS and merges it in its task queue. Subsequently, it begins to monitor the next alive successor in the circular queue. Apart from handling independent worker failures, the system also handles cascading failures (worker fails during recovery), donor failures (that cause blocking), and master failures effectively. If the master fails, the alive worker with the least sequence number is promoted to master. More details are presented in our earlier work RD-FCA [25].

4.2 System implementation

For running a job, executor processes with (*ids* 0 to *n*) are launched in the cluster. These processes participate in the master election by creating a sequential ephemeral znode in ZooKeeper. The one with the lowest sequential number becomes the master, while others act as workers. A watch ring (circular queue) is set up among the workers in the increasing order of their sequence numbers; each worker sets up a watch on its successor's znode in ZooKeeper. As each of these znodes is configured as ephemeral, when a worker fails, the znode that it created

disappears. Consequently, the watch on the predecessor worker gets triggered. As it is a ring structure, even if a sequence of workers fails together, the recovery is possible as discussed in Sect. 4.1.2 by the predecessor of the last worker that failed in the sequence.

Zookeeper's watch mechanism is also used for signaling the termination of search, completion events such as SLB completion, scaling mode changes such as scale-in/scale-out/normal mode, solution sharing, and workload estimates by workers. Certain well-defined paths in ZooKeeper are designated for these events, and the workers set up a watch on them. They get notified when these `znode`s are created or updated, for example, `znode` */SLB_Done/* is created when SLB is complete; scaling mode changes are signaled by updating the value of the `znode`. As these events occur not that frequently, sending out notifications to all the workers does not create a bottleneck. Kafka is used for implementing reliable high-throughput work transfers. Each worker creates a topic in Kafka for receiving messages from other executors (like a mailbox). During SLB, the master sends the initial splits of the problem to different workers by writing to their topic in Kafka. It is also used for task transfers from the donor to the requester.

4.3 Elastic scaling

The idea on which the elasticity controller works is monitoring the entire system to approximate runtime parallel efficiency (E_{elastic}) and vary the number of workers to keep E_{elastic} within the user-defined range (`uEff` - `dEff`). `uEff` is the up-efficiency threshold and `dEff` is the down-efficiency threshold. At the implementation level, the elasticity controller accepts a single value for the efficiency threshold and creates a range (`uEff - dEff`). Using a single threshold value for making scaling decisions would lead to thrashing. During runtime, the arithmetic average of efficiencies of individual workers gives E_{elastic}. In Sect. 2.3, we defined and explained E_{elastic} (Definition 1).

The elasticity controller of DiGTreeS uses the hybrid scaling strategy (Algorithm 3). Computation starts with 2 workers. Note that, if the initial number of workers (INW) is `N` then 1 becomes master, and the rest `N`−1 become workers (slaves). The number of workers scales out exponentially till at least one of the following two conditions is met. (1) The number of workers reaches the worker threshold ($Workers_{\text{threshold}}$), or (2) E_{elastic} reaches the desired range. We call this phase the Exponential start phase (`ES-Phase`). After the `ES-Phase`, scaling operation (up or down) is done in the granularity of 1. This is `Linear-Phase`. We use 18 workers as $Workers_{\text{threshold}}$ for the experimental evaluation of DiGTreeS. But it can be set to any number since DiGTreeS is loosely coupled in nature and most of its components are designed independently. We use 18 as the problem instances under evaluation are small and under the experimental setup, a maximum of 40 workers is possible. For the experiments with larger inputs, we have changed $Workers_{\text{threshold}}$ to 84. It has been discussed in Sect. 5.3 where we show the scalability of the proposed framework using a larger instance of GSSSA (GSSSA_L) on a test-bed that allows

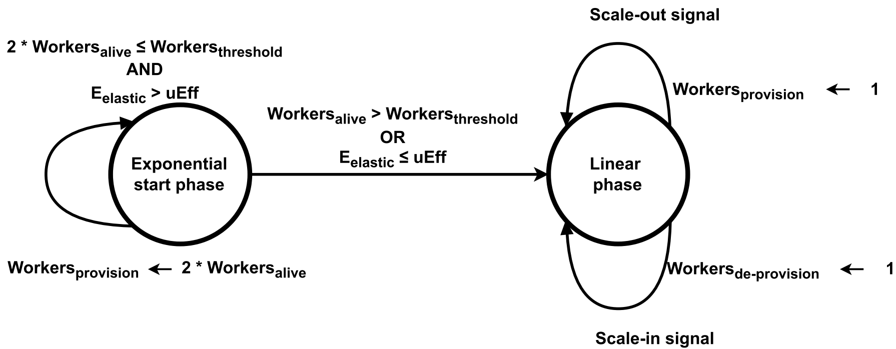


Fig. 4 Execution starts in the exponential start phase (`ES-Phase`) with two workers. The number of alive workers ($Workers_{\text{alive}}$) exponentially scales out by provisioning twice the number of currently alive workers. For the execution to remain in the `ES-Phase`, both of the following conditions are to be met: (1) $Workers_{\text{alive}} \leq Workers_{\text{threshold}}$, and (2) $E_{\text{elastic}} > \texttt{uEff}$, where $Workers_{\text{threshold}}$ is the threshold for the maximum number of workers in the `ES-Phase` (DiGTreeS uses 18 as value for $Workers_{\text{threshold}}$), E_{elastic} is the elastic efficiency, and `uEff` is the upper-efficiency threshold. If any of the two mentioned conditions gets invalidated, execution gets transitioned to the `Linear-Phase`, where scale-out and scale-in operations are done in the granularity of 1. Once `Linear-Phase` has been attended, execution remains in this phase till its completion. For Scale-out and Scale-in signals see Lines 21–23 and Line 24–26, respectively, of Algorithm 3

scaling out up to 160 workers. Figure 4 explains `ES-Phase` and `Linear-Phase` using a state transition diagram.

A prevalent problem in elasticity controllers is oscillating effects [29]. Frequent provisioning and de-provisioning of compute resources lead to colossal overhead. In a problem that needs elastic scaling, continuous monitoring of scaling metrics is required. A very small interval may be chosen for the monitoring subroutine to run, which may lead to very frequent scaling operations. Scaling operations done frequently lead to thrashing, and if the duration between two scaling operations is very large, it leads to either alive workers getting overloaded or underutilized. To address this issue, we use a heuristic interval of 7 s for monitoring the scaling metric. However, the scaling operation is carried out only when the requirements for the same scaling operation (up or down) are met thrice consecutively. Note that the biggest variations using hybrid[9] scaling seem to happen at the start and toward the end of the execution [4]. DiGTreeS addresses the variation at the start by exponentially provisioning the workers during the `ES-Phase`.

The elasticity controller (see Algorithm 3) accepts the up efficiency threshold (`uEff`) and down efficiency threshold (`dEff`) and tries to keep the E_{elastic} within these two thresholds. `decommissionCount`, `provisionCount`, `provision` and `decommission` variables are initialized (Lines 2–3). These variables help in reducing the thrashing effect. Scale-in granularity (`granIn`) and scale-out granularity (`granOut`) are set to 1. And the `startPhase` denoting `ES-Phase` is set to true (Lines 4–5). The elasticity controller computes E_{elastic} (`avgEff`) using

9 Hybrid scaling is the combination of both upscaling and downscaling.

getCPULoad method (Line 7). If the execution is still in the ES-Phase, twice the number of alive workers (aliveWorkers) is provisioned (Lines 10–14). After the ES-Phase is over, Linear-Phase starts (Lines 15–26). When E_{elastic} surpasses the up efficiency threshold (uEff) consecutively for 3 iterations, a new worker is provisioned (Lines 21–23). And when E_{elastic} falls below the down efficiency threshold (dEff) consecutively for 3 iterations, a worker is de-provisioned (Lines 24–26).

Algorithm 3 Elasticity Controller (Hybrid Scaling)

```
1:  Input: Up efficiency threshold uEff and Down efficiency threshold dEff
 :         Workers threshold Workers_threshold
2:  decommissionCount ← 0, provisionCount ← 0
3:  provision ← false, decommission ← false
4:  granIn ← 1, granOut ← 1
5:  startPhase ← true, Workers_threshold ← 18
6:  Function ELASTICITY_CONTROLLER():
7:    avgEff ← getCPULoad()
8:    aliveWorkers ← getAliveWorkersCount()
9:    n ← aliveWorkers * 2
10:    if avgEff > uEff and n ≤ Workers_threshold and startPhase then
11:       granOut ← n
12:    else
13:       granOut ← 1
14:       startPhase ← false
15:    if avgEff > uEff then
16:       provison ← true, decommissionCount ← 0
17:       provisionCount++
18:    else if avgEff < dEff then
19:       decommission ← true, provisionCount ← 0
20:       decommissionCount++
21:    if provision and provisionCount ≥ 3 then
22:       increaseWorkers(granOut), provisionCount ← 0
23:       provision ← false
24:    if decommission and decommissionCount ≥ 3
25:       decreaseWorkers(granIn), decommissionCount ← 0
26:       decommission ← false
```

When a worker is provisioned, it asks the master for the donor worker's id and then directly requests the donor worker for task transfer (the task transfer mechanism is discussed in detail in Sect. 4.1.1). DiGTreeS does not drop a worker when it has pending local tasks because migrating work from a busy worker will be overhead. At the implementation level, when a drop signal arises from the elasticity controller a corresponding znode is created in ZooKeeper. When a worker completes its local task, before requesting more work, it checks for the presence of the said znode. If it is present, the worker terminates itself, leading to worker drop. Checking for znode and drop operation is an atomic action.

5 Experimental evaluation

All experiments have been conducted on a setup comprising server-class machines, with a dual 10-core Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz CPU, 128 GB RAM, and running Ubuntu 16.04.3 LTS and Java v11.0.10. Hadoop v3.2.0, ZooKeeper v3.4.1, and Kafka v2.2.0 are also installed on these machines. Unless mentioned otherwise, each experiment has been performed at least 7 times, and stats about deviation[10] from the mean values of execution time and cost for all cases are discussed. Since all experiments are carried out under a controlled environment, we see very minimal deviations.

The following datasets have been used in the experiments: (1) Travelling Salesman Problem: the test cases are generated using a tool[11] that takes as input the number of cities and a seed value. We use a TSP instance of 35 cities with a seed value of 37662. (2) 0–1 Knapsack Problem (Knap): We used the generator described in [30] to generate hard instances with the configuration setting Type = uncorrelated span(2, 10), TestNo = 28. (3) N-queens problem (NQ): the number of queens is specified as 16. (4) Generic State Space Search Application problem (GSSSA): We use the problem instance described in [7]; its parameter values are $W_r = 1 \times 10^9$, $W_i = 19 \times 10^9$, $b = 2 \times 10^{-4}$, and $g = 1 \times 10^6$. We also use a larger instance of GSSSA (GSSSA_L) with input parameters $W_r = 1 \times 10^{20}$ and $W_i = 19 \times 10^{20}$ with b and g the same as above to demonstrate the scalability of DiGTreeS. Results regarding cost and execution time for the N-queens (NQ) are shown separately because it has a relatively shorter execution time.

5.1 Performance with a fixed number of workers

Here a fixed number of workers are used for the entire search. From Fig. 5a, we see that the time taken to complete the search reduces with an increase in the number of workers. But this also has a trade-off in terms of the efficiency of computation (Fig. 5b). The efficiency with N workers (which is calculated using Eq. 2) starts to drop as the number of workers increases. In these experiments, we observe that beyond 20 workers, the efficiency is below 0.5. Across different problems, the efficiency at a higher number of workers is highest for N-queens, then TSP then Knap, and GSSSA.

GSSSA allows us to control the irregularity by varying the balancing factor b. Along with the GSSSA with input parameters described in the previous paragraph and [7], we run a special case of GSSSA with very high irregularity b = 0.00001 and call it GSSSA_S. This extreme irregularity is the reason for very little speedup with the increase in the number of workers in the case of GSSSA_S (see Fig. 5a). This extreme irregularity is highly unlikely to be present in a

[10] We calculate the percentage deviation as the ratio of difference of current value and mean value to the mean value, i.e., deviation = $\frac{|\text{current value} - \text{mean value}|}{\text{mean value}} \times 100$.

[11] http://dimacs.rutgers.edu/archive/Challenges/TSP/.

real-world problem. We further analyze the performance of DiGTreeS using weak scaling experiments.

Weak scaling is how the execution time varies with the number of workers for a fixed problem size per worker. The problem size increases at the same rate as the number of workers, keeping the amount of work per worker the same. To test DiGTreeS' performance using weak scaling we use the GSSSA_L instance. We start execution with parameters, $W_r = 1 \times 10^{20}$, $W_i = 19 \times 10^{20}$, and number of workers = 120. Balancing factor b and granularity g remain the same as described earlier. Then for each iteration, we reduce W_r, W_i, and the number of workers by half and record the execution time for each case. From Table 1, we see that execution time does not vary much as the problem size per worker remains the same. This shows that DiGTreeS is scalable as the communication overhead is minimal as we increase the number of workers.

5.2 Analysis of the hybrid scaling strategy in DiGTreeS

Here, we evaluate the hybrid scaling strategy in terms of execution time, and the total cost (Eq. 4) incurred. We compare its performance with 3 different baseline strategies: fixed number of workers, up-scaling, and down-scaling. We show results for GSSSA, TSP, NQ, and Knap. For up-scaling and hybrid scaling, we start with a smaller number of workers (INW = 2), whereas for down-scaling we start with 30 workers. Values for `uEff` and `dEff` are set to 0.95 and 0.85, respectively. Scale-out and scale-in granularity is set to 1.

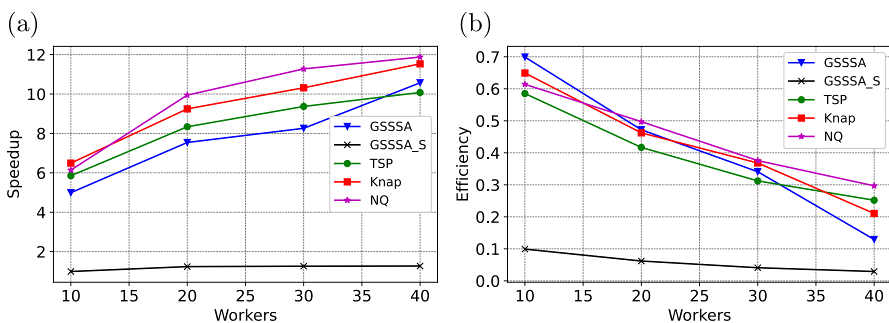


Fig. 5 Variation of **a** speedup and **b** efficiency with the static number of workers for different applications

Table 1 Execution time during weak scaling for GSSSA_L

S. No.	$W_r = 1 \times 10^{20}$	$W_i = 19 \times 10^{20}$	Workers	Execution time (s)
1	W_r	W_i	120	15852
2	$W_r/2$	$W_i/2$	60	15501
3	$W_r/4$	$W_i/4$	30	15004
4	$W_r/8$	$W_i/8$	15	14789

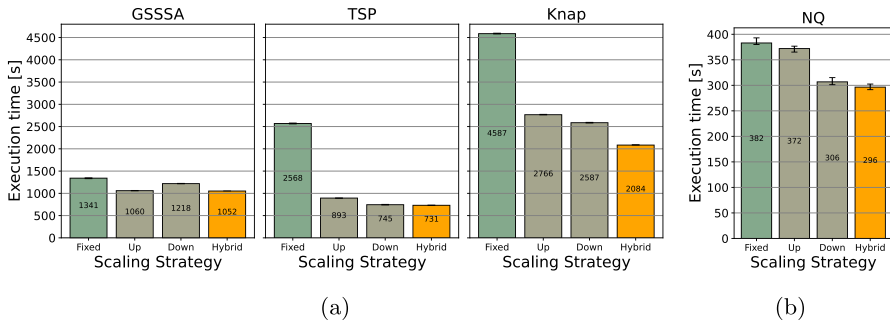


Fig. 6 Execution time for different scaling strategies

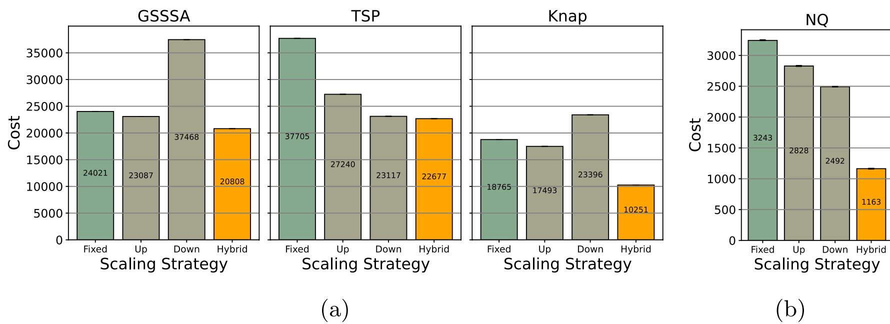


Fig. 7 Cost for different scaling strategies

In terms of execution time, hybrid scaling proves to be the best strategy for all four application problems. Recall that (see Sect. 2.6), a decision problem is a special case of optimization and enumeration problems. We showed results for optimization and enumeration problems only. These results hold for all three classes (enumeration, decision, and optimization). From Fig. 6 we see that the execution time for hybrid scaling is always lower than all the other approaches. Hybrid scaling switches between up-scaling and down-scaling to minimize the overhead of parallel execution. Considering execution times across all seven runs we get a maximum deviation of 3.7%.

In terms of cost (which is calculated using Eq. 4), the hybrid scaling strategy performs better than the other strategies (see Fig. 7) as well. No extra cost is incurred by needlessly keeping a worker around or by working with fewer workers for a long time. Across all runs, we get a maximum deviation of 1.0%. Thus, hybrid scaling proves to be the best strategy for all four problems in terms of both execution time and cost.

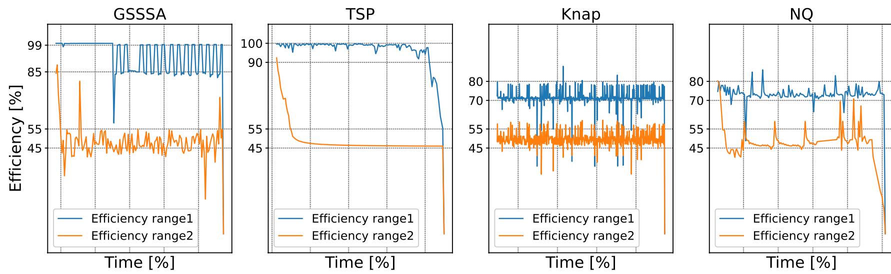


Fig. 8 Efficiency curve with elastic scaling for different efficiency ranges. For GSSSA efficiency range1 is (99%, 85%) and efficiency range 2 is (55%, 45%). Similarly, for other problem instances, both efficiency ranges are highlighted on the *y*-axis

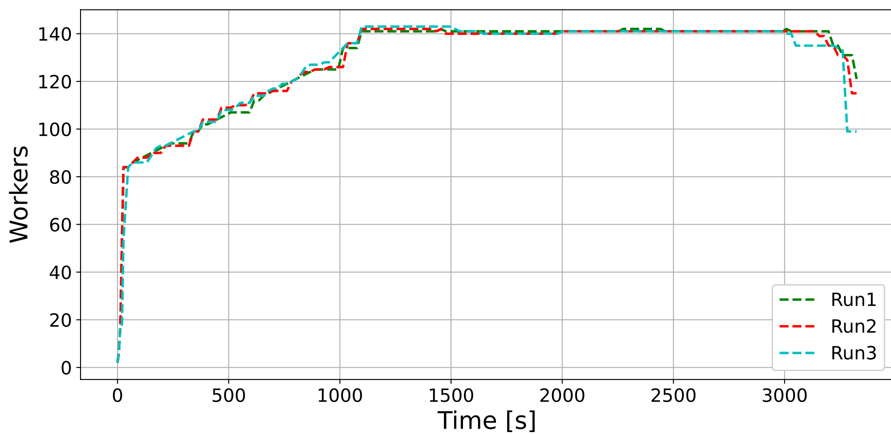


Fig. 9 Workers employed as a function of time for GSSSA_L instance. For all three runs, the elasticity controller shows stable behavior

5.3 Maintaining efficiency within user-defined range

We discussed elastic scaling in Sect. 4.3. Here, we test the ability of the elasticity controller (which uses elastic scaling based on the hybrid scaling strategy) of DiGTreeS to maintain E_{elastic} within a range (`dEff` - `uEff`). The elasticity controller is able to keep the E_{elastic} within any defined range. To demonstrate this we show the results for all four problem instances with two different efficiency ranges. $\texttt{uEff}_1 = 0.99$ and $\texttt{dEff}_1 = 0.85$ for GSSSA; for TSP, $\texttt{uEff}_1 = 0.99$ and $\texttt{dEff}_1 = 0.90$; for NQ, $\texttt{uEff}_1 = 0.80$ and $\texttt{dEff}_1 = 0.70$; and for Knap, $\texttt{uEff}_1 = 0.80$ and $\texttt{dEff}_1 = 0.70$ as well. $\texttt{uEff}_2$ and $\texttt{dEff}_2$ are set to 0.55 and 0.45, respectively, for all problem instances.

From Fig. 8, we see that the elasticity controller maintains the (E_{elastic}) within the defined ranges. During the `ES-Phase`, E_{elastic} is always high as the number of workers is less. Because of the irregular nature of the problem instances, there are

Table 2 Execution time and cost incurred by DiGTreeS compared to Helpar [8] and equilibrium [4]

	Execution time (s)			Cost (Eq. 4)		
	Helpar	Equilibrium	DiGTreeS	Helpar	Equilibrium	DiGTreeS
TSP	2264	857	915	32815	23961	18700
GSSSA	1275	1476	1260	16374	14420	14967
Knap	3570	3544	3500	26046	10502	10327
NQ	370	384	370	1859	1443	1150

frequent fluctuations in the efficiency curves, but they remain within the range for the majority of the time during execution.

To further understand the E_{elastic} fluctuations' impact on the scalability of the system we performed additional experiments with larger instances using GSSSA_L on a larger test-bed where the number of workers can go up to 160. Figure 9 shows such a set of runs using GSSSA_L. Since the input size is relatively larger, we have used $Workers_{\text{threshold}}$ as 84 so that E_{elastic} reaches the desired efficiency range quickly and the efficiency range is set to `uEff` = 0.70 and `dEff` = 0.60. For all three runs, the elasticity controller shows stable behavior and E_{elastic} fluctuations do not result in the thrashing of workers during the execution. Moreover, while testing DiGTreeS for scaling using GSSSA_S (a special case of GSSSA with extreme irregularity), with repeated runs we observed that the elasticity controller does not provision more workers (in this case a maximum of 4 workers got provisioned) when the problem is not scalable. The elasticity controller can detect the scaling behavior of the problem.

5.4 Comparison with the state-of-the-art elastic controllers

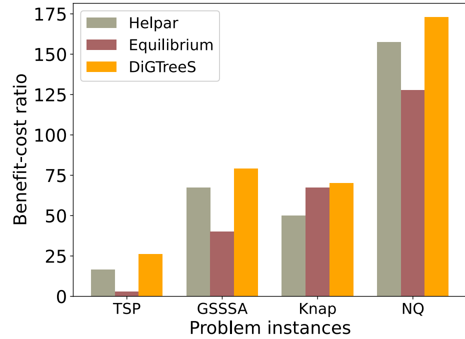
Equilibrium [4] and Helpar [8] are the most recent works in elastic scaling for parallel tree search. We implemented elasticity controllers of Equilibrium [4] and Helpar [8] and compared the elasticity controllers of DiGTreeS with them. From Table 2 we see that DiGTreeS performs better than Helpar and Equilibrium both in terms of execution time as well as cost in most of the cases. In some cases, we see comparable results. For instance, Equilibrium has a lower execution time for TSP than DiGTreeS, but the cost is higher. So, we further perform a benefit–cost analysis.

Benefit–cost ratio [31] (BCR) is an indicator used in cost–benefit analysis. Originally, BCR is calculated by the following equation:

$$BCR = \frac{\text{present value of expected benefits}}{\text{present value of expected costs}} \tag{6}$$

BCR gives the benefit obtained per unit cost incurred. The higher the value of BCR, the better the approach. We consider the execution time and cost with a fixed number of workers (30) as the baseline. The benefits derived by elastic scaling are computed

Fig. 10 Performance of DiGTreeS vs state of the art [4, 8]



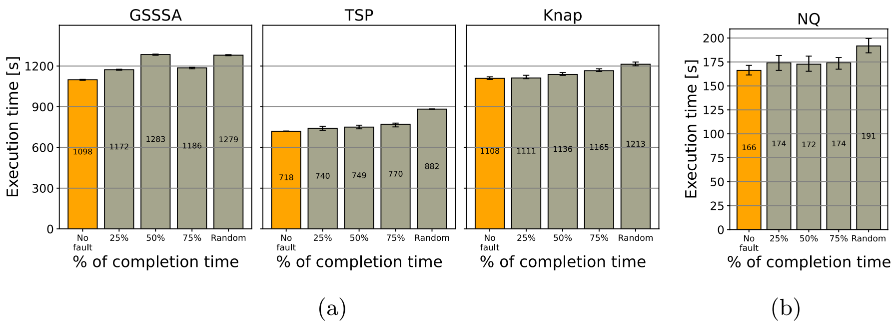


Fig. 11 Master failure at different instances during execution

as the reduction in the monetary cost w.r.t. baseline, while the cost is measured in terms of an increase in the execution time w.r.t. baseline. We observe that DiGTreeS outperforms both the state-of-the-art elastic controllers in all the problem instances (see Fig. 10).

5.5 Performance of DiGTreeS under failure

As discussed in Sect. 1, the tree searches are long-running jobs; without fault tolerance, the entire job will have to be rerun until successful completion. Resilience to failures is one of the three desirable properties of a cloud-aware framework. Here we study the performance of DiGTreeS under various failure scenarios. To emulate the performance of DiGTreeS under different failure scenarios we implemented the failure injection mechanism used in our earlier work RD-FCA [25].

The fault injection subroutine creates `znode` corresponding to the specified failure. Each worker checks for the presence of failure `znode`, and if found, tries to delete it. If the deletion is successful, the current worker fails by self-termination. In case of multiple worker failures, groups of workers are formed equal to the number

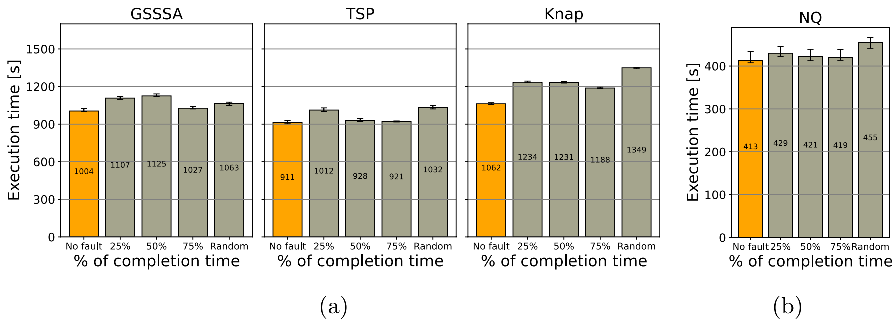


Fig. 12 Execution time for single-worker failure at different instances during execution

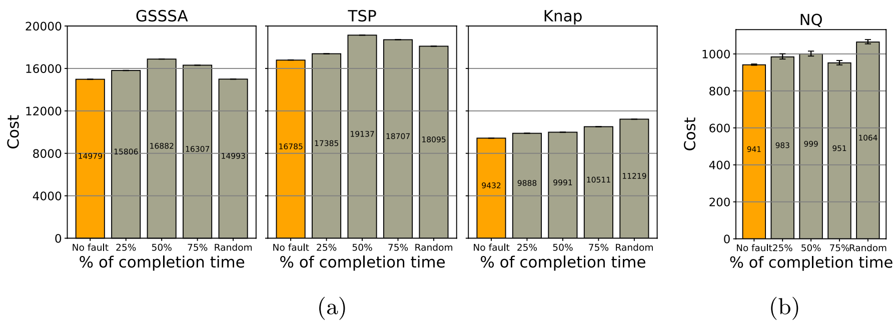


Fig. 13 Cost (Eq. 4) of single-worker failure at different instances during execution

of failures. Workers from each group attempt deletion of the corresponding failure `znode` and only one of them succeeds. For further details refer RD-FCA [25].

5.5.1 Performance under master failure

We analyze the effect of master failure on completion time by introducing master failure at different phases (at 25%, 50%, 75%, and at a random time) during the execution of different problem instances. Master failure at any instance leads to increased execution time, but the increment is always less than 10% of the time taken in the no-fault scenario (see Fig. 11). With repeated runs, the deviation is stable with the maximum deviation of 9.0% observed in NQ when the failure is introduced at a random point of execution. A failure of the master does not impact execution much. Recall that (from Sect. 4.1.2) when the master fails a worker is promoted to master. It takes over the responsibilities of the failed master. Loss of worker is adjusted by the scaling algorithm (Algorithm 3). Details about the effect of loss of workers are presented in Sect. 5.5.3 where we discuss the impact of multi-worker failure.

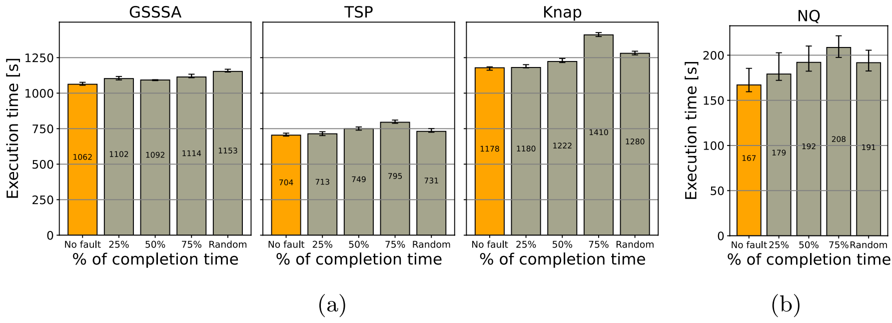


Fig. 14 Execution time for multiple workers failure at different instances during execution

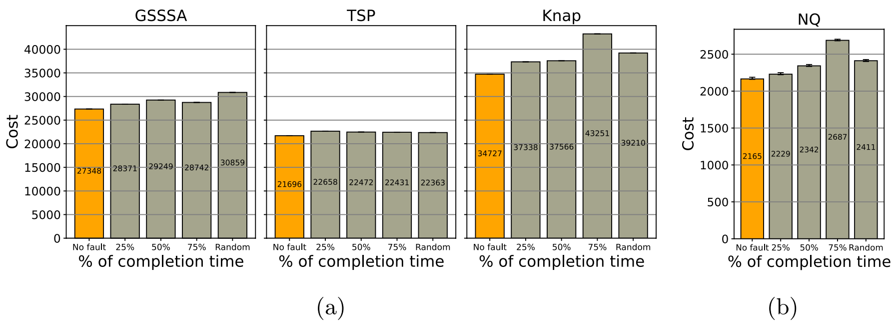


Fig. 15 Cost (Eq. 4) of multiple worker failure at different instances during execution

5.5.2 Performance under single-worker failure

To evaluate the performance of DiGTreeS under single-worker failure, we introduce the worker failure at different instances (at 25%, 50%, 75%, and at random time) during the execution and compare execution time and cost with the no-fault scenario.

From Figs. 12 and 13, we observe that in most of the single-worker failure scenarios, the execution time and the cost match the no-fault scenario. With repeated runs, the deviation is stable with the maximum deviation of 6.5% and 2.8% observed in execution time and cost, respectively, for NQ. Thus, we have very little recovery overhead. To further see the impact of failure we analyze execution time and cost with multiple worker failures.

5.5.3 Performance under multiple worker failure

To evaluate the performance of DiGTreeS in the presence of multiple worker failures, execution starts with 10 workers.[12] We introduced 4 worker failures at different instances during the execution.

[12] Execution starts with 10 workers as workers may fail at the very beginning of the execution when the number of workers is small (i.e., less than 4).

Table 3 Ratio of cost to execution time in different failure scenarios for multiple worker failure

Problem instance	No fault	25%	50%	75%
GSSSA	25.93	25.94	26.92	25.93
TSP	29.40	32.30	32.27	28.58
NQ	13.73	12.89	12.82	13.76
Knap	30.03	32.19	30.91	30.94

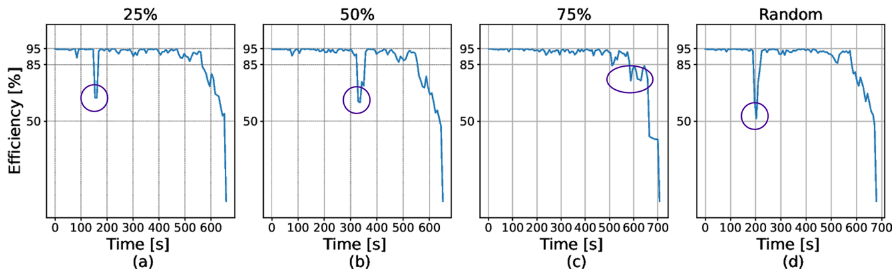


Fig. 16 Efficiency curve for TSP under multiple worker failure at different instances during the execution. We see a dip in the E_{elastic} when multiple workers fail simultaneously

From Figs. 14 and 15, we see that the execution time and cost for all the failure scenarios are comparable to the no-fault scenario. Among GSSSA, TSP, and Knap with repeated runs, we get a maximum deviation of 3.8% and 0.1% in execution time and cost, respectively. For NQ we get a maximum deviation of 16.0% in execution time. This is a general observation. Among all four problem instances, N-queens have the least execution time and cost and it shows the most deviations. As the problem instance gets bigger, behavior gets stable and deviations get lesser.

To analyze this further we compute the cost to execution time ratio. From Table 3, we see that the ratios of cost to execution time in different failure scenarios for multiple worker failure are very close to the no-fault scenario. It means the cost incurred per unit execution time is very close to the no-fault scenario for all the problem instances in most of the cases, i.e., DiGTreeS can give a similar cost for unit execution time irrespective of whether there is worker failure or not. This points to a positive justification of the recovery mechanism used in DiGTreeS. It shows the effectiveness of the fault tolerance mechanism of DiGTreeS.

When a worker fails, the monitoring worker reads the latest snapshot of the failed worker from HDFS and merges it in its task queue (for more details on fault tolerance see Sect. 4.1.2). This merging of the failed worker's task queue to the monitoring worker's task queue does not contribute toward essential computations and thus leads to a decrease in E_{elastic}. This drop in E_{elastic} is quickly addressed by the fault-tolerance mechanism of DiGTreeS, and the elasticity controller (Algorithm 3) pulls the efficiency back to the desired range. Figure 16 shows the variation in the $E_{elastic}$ for 4 independent worker failures at different execution points for the TSP problem instance. From Table 3, we can see that the cost-to-execution time ratio does not deviate much for all the cases.

5.6 Summary of the results

Through experiments and discussion, we showed that DiGTreeS is scalable using weak scaling. The hybrid scaling strategy is best suited for the pay-per-use model. With two different ranges we saw DiGTreeS can maintain E_{elastic} within any given range and with larger inputs (see Fig. 9) we observed that DiGTreeS shows a stable behavior. Comparison with the state-of-the-art elasticity controller revealed that DiGTreeS outperforms them in terms of BCR. Further, we saw that DiGTreeS can recover various failure scenarios with minimal overheads. In our earlier work [25], we showed that snapshots used in checkpointing for fault tolerance have an overhead of less than 5%. Experiments till now were based on the reactive approach. Further, we discuss a proactive approach.

6 Future directions: proactive approach

We have seen the reactive approach. Recall our discussion from Sect. 4.3 about `ES-Phase` and `Linear-Phase`. We saw how DiGTreeS takes care of variations due to hybrid scaling at the start of execution by exponentially provisioning the workers during the `ES-Phase`. Similar variations occur toward the end of execution as well. Here, we see the scope for using the proactive approach[13] (also known as the predictive approach). As the set of problems under consideration is irregular, due to pruning of a subtree E_{elastic} may vary considerably. In the reactive approach, once `Linear-phase` is attained, it may take a considerable amount of time to re-attain the desired E_{elastic}. This is another scenario where we see the scope for using the proactive approach. In the proactive approach, the elasticity controller can anticipate the required number of workers to scale out (or scale in) to reach the desired E_{elastic} and provision (or de-provision) workers at once instead of scaling in multiple of one. This would greatly reduce the execution time.

The proposed proactive approach attempts to create a *WHAT-IF* engine using machine learning techniques to predict the number of workers to be scaled in solving an irregular tree search problem while maintaining E_{elastic} as close as possible to a desired value.

There are certain challenges in building such an engine: the irregular nature of the problem; for our initial study we chose variables like etc. and used supervised machine learning techniques there must be a pattern in the problem. Due to the irregularity, the chances of finding any pattern are very low. However, variables like type of the problem (*viz.* optimization, enumeration or decision), progress (number of nodes explored), number of alive workers, and E_{elastic} can be correlated. Given a type of problem, the number of alive workers and E_{elastic} is inversely proportional. During the initial phase when the progress is less, the number of scaling operations is high. We can see patterns in these variables.

[13] It is the technique to anticipate future changes in the system and act accordingly before it occurs [6].

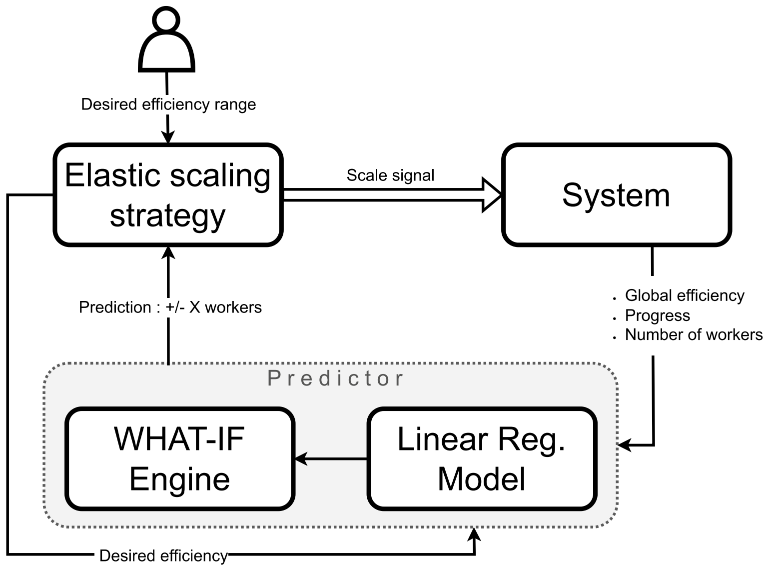


Fig. 17 High-level working of proactive scaling strategy

The proposed methodology uses supervised learning techniques to train the machine learning model. The variables to train the model are identified and collected as a first step, which are then used to train a Linear Regression Model. The scaling strategy uses this model to predict the new efficiency upon the addition or deletion of workers. The entire methodology can be divided into 3 major units: (1) Monitor, (2) Predictor, and (3) Decision maker. The monitoring unit collects data about the state of the machine including the amount of work already done (progress), the current number of alive workers, global efficiencies, etc. The predictor uses the same to predict the new efficiency upon the addition/deletion of X number of workers, using the model trained earlier. Finally, the decision-maker unit evaluates the required number of workers such that the new efficiency comes as close as possible to the desired value (same can be inferred from Fig. 17).

6.1 Data collection and model training

We have used four independent variables: (1) Progress in the search space, (2) E_{elastic}, (3) Current Number of alive workers, and (4) Workers to be scaled. Due to the addition/deletion of workers, the efficiency changes. This is the dependent variable and is logged after intervals in multiples of 30 s. It is important to note that the number of workers to be scaled can be negative to signify that the workers are dropped.

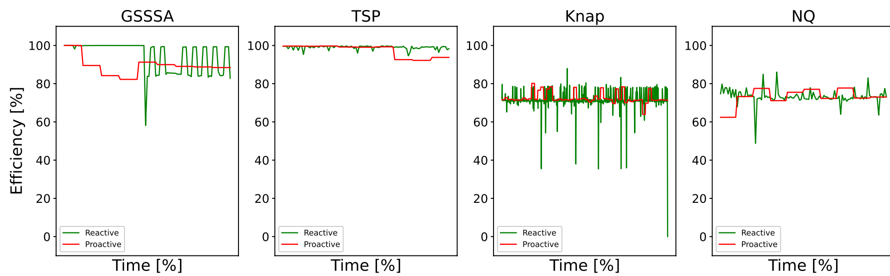


Fig. 18 Efficiency with proactive hybrid scaling

To train the Linear Regression Model (see Eq. 7), we gathered data points in intervals of 30 s by running the reactive hybrid scaling algorithm on smaller instances of the application problem. These data points are split into two sets, training and testing having a weightage of 70% and 30%, respectively. The trained model had a regression score of 0.94 for test data and 0.96 for training data.

$$E_{\text{pred}} = \mathrm{A} \cdot E_{\text{elastic}} + \mathrm{B} \cdot W_{\text{alive}} + \mathrm{C} \cdot N_{\text{exp}} + \mathrm{D} \cdot G_{\text{scale}} \tag{7}$$

where E_{elastic} is the current elastic efficiency, W_{alive} is the number of alive workers, N_{exp}, is progress in the search space, and G_{scale} denotes the number of workers to be scaled out/in.

Recall our discussion from Sect. 5.3, where we observed fluctuation in the E_{elastic} curve. The proposed proactive scaling strategy roughly scales the exact number of workers required to reach a particular desired efficiency. This leads to the addition of workers more swiftly and convergence quickly. Earlier, we observed that the efficiency curve for reactive hybrid scaling showed fluctuations due to the irregular search tree. In the proactive approach, the frequency of fluctuations has reduced as the worker(s) is/are provisioned/de-provisioned in advance based on the predicted efficiency (see Fig. 18). Furthermore, we note that the proactive model is also capable of keeping the computational efficiency within the user-defined range.

Currently, the proactive model of DiGTreeS is in the initial stages and more work and extensive testing is required. We plan to do this in our future work. The model is trained only on the data of a specific problem at a time and hence lacks generality. A new independent variable of problem type (enumeration, optimization, and decision) can be introduced further to account for problem-specific patterns such that the model can be used for more general problems.

7 Conclusion

In this paper, we presented the design and implementation of DiGTreeS, a distributed framework that supports generalized exact combinatorial search. We implemented 4 search problems and showed that the proposed elasticity controller performs well with the varying workload and efficiency and elastic scaling can be

beneficial in reducing the search time and cost compared to a fixed number of workers. We implemented an elasticity controller based on the hybrid scaling strategy. Comparison with state of the art [4, 8] revealed that DiGTreeS performs better in terms of benefit–cost ratio (BCR). The proposed framework not only outperforms the existing elasticity controllers in terms of BCR but also provides all the three desirable properties of a cloud-aware framework for parallel combinatorial search viz. (1) ease of expressing parallelism, (2) resilience to failures, and (3) support for elastic scaling.

Though the proactive approach looks good, it is still in its initial phase and needs further improvements. Currently, a separate model needs to be trained for different categories (optimization, decision, and enumeration) of the problem. There can be a single model with a separate parameter for the category of the problem. Extensive testing of the proposed proactive model under different failure scenarios is also required. These are the parts of our future work.

Author contribution MAJ helped in conceptualization, methodology, writing—original draft, writing—reviewing and editing, software. SK contributed to conceptualization, methodology, writing—reviewing and editing. BG and VS were involved in conceptualization and methodology.

Data availability Data and/or code will be made available on request.

Declarations

Conflict of interest The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Paschos VT (2014) Applications of combinatorial optimization. Wiley, Hoboken
2. Archibald B, Maier P, Stewart R, Trinder P (2019) Implementing yewpar: a framework for parallel tree search. In: Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25. Springer, pp 184–196
3. Goldreich O (2010) P, NP, and NP-completeness: the basics of computational complexity. Cambridge University Press, Cambridge
4. Kehrer S, Blochinger W (2020) Equilibrium: an elasticity controller for parallel tree search in the cloud. J Supercomput 76:9211–9245
5. Yasugi M, Muraoka D, Hiraishi T, Umatani S, Emoto K (2019) Hope: a parallel execution model based on hierarchical omission. In: Proceedings of the 48th International Conference on Parallel Processing, pp 1–11
6. Rampérez V, Soriano J, Lizcano D, Lara JA (2021) Flas: a combination of proactive and reactive auto-scaling architecture for distributed services. Futur Gener Comput Syst 118:56–72
7. Haussmann J, Blochinger W, Kuechlin W (2019) Cost-efficient parallel processing of irregularly structured problems in cloud computing environments. Clust Comput 22(3):887–909
8. Rosa Righi R, Rodrigues VF, Rostirolla G, Costa CA, Roloff E, Navaux POA (2018) A lightweight plug-and-play elasticity service for self-organizing resource provisioning on parallel applications. Futur Gener Comput Syst 78:176–190

9. Vizel Y, Weissenbacher G, Malik S (2015) Boolean satisfiability solvers and their applications in model checking. Proc IEEE 103(11):2021–2035
10. Yang J, He Q (2018) Scheduling parallel computations by work stealing: a survey. Int J Parallel Prog 46:173–197
11. Xie F, Davenport A (2010) Massively parallel constraint programming for supercomputers: Challenges and initial results. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming. Springer, pp 334–338
12. Herbst NR, Kounev S, Reussner R (2013) Elasticity in cloud computing: what it is, and what it is not. In: 10th International Conference on Autonomic Computing (ICAC 13), pp 23–27
13. Hunt P, Konar M, Junqueira FP, Reed B (2010) Zookeeper: wait-free coordination for internet-scale systems. In: USENIX Annual Technical Conference, vol 8
14. Kreps J, Narkhede N, Rao J et al. (2011) Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, vol 11. Athens, Greece, pp 1–7
15. Shvachko K, Kuang H, Radia S, Chansler R (2010) The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp 1–10. https://doi.org/10.1109/MSST.2010.5496972
16. Gupta A, Faraboschi P, Gioachin F, Kale LV, Kaufmann R, Lee B-S, March V, Milojicic D, Suen CH (2014) Evaluating and improving the performance and scheduling of hpc applications in cloud. IEEE Trans Cloud Comput 4(3):307–321
17. Bui P, Rajan D, Abdul-Wahid B, Izaguirre J, Thain D (2011) Work queue+ python: a framework for scalable scientific ensemble applications. In: Workshop on Python for High Performance and Scientific Computing at Sc11
18. Rosa Righi R, Rodrigues VF, Da Costa CA, Galante G, De Bona LCE, Ferreto T (2015) Autoelastic: automatic resource elasticity for high performance applications in the cloud. IEEE Trans Cloud Comput 4(1):6–19
19. Archibald B, Maier P, Stewart R, Trinder P, De Beule J (2017) Towards generic scalable parallel combinatorial search. In: Proceedings of the International Workshop on Parallel Symbolic Computation, pp 1–10
20. Poldner M, Kuchen H (2008) Algorithmic skeletons for branch and bound. In: Software and Data Technologies: First International Conference, ICSOFT 2006, Setúbal, Portugal, September 11–14, 2006, Revised Selected Papers 1. Springer, pp 204–219
21. Bungart M, Fohry C (2017) A malleable and fault-tolerant task pool framework for x10. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp 749–757
22. Johnson DS, McGeoch LA (1997) The traveling salesman problem: a case study in local optimization. Local Search Comb Optim 1(1):215–310
23. Salkin HM, De Kluyver CA (1975) The knapsack problem: a survey. Naval Res Logist Q 22(1):127–144
24. Bell J, Stevens B (2009) A survey of known results and research areas for n-queens. Discret Math 309(1):1–31
25. Khaund A, Sharma AM, Tiwari A, Garg S, Kailasam S (2023) Rd-fca: a resilient distributed framework for formal concept analysis. J Parall Distrib Comput 179:104710
26. Archibald B, Maier P, McCreesh C, Stewart R, Trinder P (2018) Replicable parallel branch and bound search. J Parall Distrib Comput 113:92–114
27. Prim RC (1957) Shortest connection networks and some generalizations. Bell Syst Tech J 36(6):1389–1401
28. Kizilateş G, Nuriyeva F (2013) On the nearest neighbor algorithms for the traveling salesman problem. In: Advances in Computational Science, Engineering and Information Technology: Proceedings of the Third International Conference on Computational Science, Engineering and Information Technology (CCSEIT-2013), KTO Karatay University, June 7–9, 2013, Konya, Turkey-Volume 1. Springer, pp 111–118
29. Bersani MM, Bianculli D, Dustdar S, Gambi A, Ghezzi C, Krstić S (2014) Towards the formalization of properties of cloud-based elastic systems. In: Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems, pp 38–47

30. David P (2005) Where are the hard knapsack problems? Comput Oper Res 32(9):2271–2284
31. Zangeneh A, Jadid S, Rahimi-Kian A (2010) Normal boundary intersection and benefit-cost ratio for distributed generation planning. Eur Trans Electr Power 20(2):97–113

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.