Cloud-based Dynamic Distributed Optimisation of Integrated Process Planning and Scheduling in Smart Factories

Shuai Zhao
Department of Computer Science,
University of York
York, UK
shuai.zhao@york.ac.uk

Piotr Dziurzanski
Department of Computer Science,
University of York
York, UK
piotr.dziurzanski@york.ac.uk

Michal Przewozniczek
Department of Computer Science,
University of York
York, UK
michal.przewozniczek@york.ac.uk

Marcin Komarnicki
Department of Comp. Intelligence,
Wroclaw University of Technology
Wroclaw, Poland
marcin.komarnicki@pwr.edu.pl

Leandro Soares Indrusiak
Department of Computer Science,
University of York
York, UK
leandro.indrusiak@york.ac.uk

ABSTRACT

In smart factories, process planning and scheduling need to be performed every time a new manufacturing order is received or a factory state change has been detected. A new plan and schedule need to be determined quickly to increase the responsiveness of the factory and enlarge its profit. Simultaneous optimisation of manufacturing process planning and scheduling leads to better results than a traditional sequential approach but is computationally more expensive and thus difficult to be applied to real-world manufacturing scenarios. In this paper, a working approach for cloud-based distributed optimisation of process planning and scheduling is presented. It executes a multi-objective genetic algorithm on multiple subpopulations (islands). The number of islands is automatically decided based on the current optimisation state. A number of test cases based on two real-world manufacturing scenarios are used to show the applicability of the proposed solution.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Applied computing** → **Industry and manufacturing**; • **Computing methodologies** → *Genetic algorithms*;

KEYWORDS

Integrated process planning and scheduling, Multi-objective Genetic Algorithm, Function as a Service, Serverless Clouds.

1 INTRODUCTION

To remain competitive in a market, smart factories need to be adapted to small batches and highly customised manufacturing [15]. These new conditions require dynamic adaptivity of the factory with regards to process planning and scheduling, governed by enterprise resource planning (ERP) and manufacturing execution systems (MES) connected to smart devices (things) [14]. An integrated process planning and scheduling is triggered after receiving a new manufacturing order or when a smart factory state change has been detected, caused by e.g. a thing failure [3]. Typically, the current factory state is inferred based on the values read from individual things and sent to an optimisation engine together with the new manufacturing order to be allocated and scheduled [1]. The final solution is then sent to the users and/or applied automatically to things [7, 14].

The optimisation process, usually quite computationally expensive, is executed on demand at time points difficult to predict. These busy intervals are followed with intervals with no significant computational demands. Such on-and-off workloads are suitable for public clouds, especially for serverless execution (aka Function as a Service, FaaS) [3, 15]. FaaS allows a cloud to run a code without prior server provisioning or managing. The computational power scales automatically, is highly available and fault tolerant. The first request can initially see several seconds response time, but is shorter than 1s for subsequent requests[1]. Using such services is also economically beneficial as there is no charge for the time when a code is not running. As FaaS automatically provides as much capacity as needed and a user is billed based on per-second capacity consumption and executions, it is suitable for performing distributed computation with several computing nodes, especially when the number of these nodes changes significantly during the execution. An example of such computation can be the Island Model of Genetic Algorithms (GAs) in which each node hosts a separate subpopulation to preserve the genetic diversity of the entire population. The islands exchange some individuals periodically. Typically, the number of islands is fixed [2], but thanks to serverless execution, it can be simply scaled up and down based on the current state of the optimisation process. Such a scheme of the optimisation can lead to better results in a shorter time. These features make it suitable for the process planning and scheduling investigated in this paper.

The proposed method can be applied to a wide range of smart factories. In particular, it is applicable to both production of distinct items (i.e. discrete manufacturing) and production using formulations or recipes (i.e. process manufacturing). Both these manufacturing branches are exemplified with the real-world use cases provided in this paper. The industrial partners that formulated these scenarios have

[1] https://console.bluemix.net/docs/openwhisk/openwhisk_compare.html

already implemented pilot solutions of the proposed method and confirmed its advantages.

The main contribution of this paper can be summarised with the following points:

- proposing two algorithms for determining the number of islands for multi-objective GAs based on the temporal state of the optimisation process,
- describing a serverless deployment of the Island Model of GAs that enhances its scalability,
- presenting two real-world scenarios formulated by manufacturing companies and their experimental evaluation.

The rest of this paper is organised as follows. After the brief survey of related works in Section 2, the general architecture of the developed system is presented in Section 3. The cloud deployment of the optimisation module is sketched in Section 4. The strategies for determining the number of islands are proposed in Section 5 and applied to real-world use cases in Section 6. Finally, the paper is concluded in Section 7.

2 RELATED WORK

GA-based optimisation in clouds has been attracting increasing interest from researchers since publication by Di Martino et al. [2]. In that publication, three parallel architectures were proposed which were performed at the fitness evaluation, population and individual levels, respectively. All three architectures applied the popular Map/Reduce programming model. However, the presented proof-of-concept implementation employed parallelism at the fitness-evaluation level only. Consequently, the communication overhead was significant. In the proposed solution, the population-level parallelism (Island Model) is employed instead.

The fine-grained level of parallelism has been also applied to a conceptual workflow in positional paper [13]. In contrast to [2], the container technology (Docker[2]) has been used rather than Google App Engine web framework. That technology has facilitated large and scalable deployments on a different infrastructure, focusing on security, consistency and reliability. In the proposed solution, Docker containerisation is also performed, but the containers are executed in a serverless fashion that results in dynamic scalability rather than setting a fixed number of containers with an optimisation engine.

Population-level parallelisation has been employed in Ma et al. [11]. The master node has assigned the individuals from each generation to the slave nodes based on their load information and then collected the corresponding fitness values. The same number of individuals has been sent to a fixed number of slaves (32, 48 and 64 nodes), where each slave maintained one island. However, selecting an appropriate fixed number of islands is difficult and hence the methods that employ Strategies of Dynamic Subpopulation Number Control (SDSNC) have been proposed. During their run, the subpopulations are created and deleted, depending on the method state. The idea is to increase the number of subpopulations when the method is stuck and to decrease it when the best-found solution is frequently improved. The examples of such propositions are Classic [8] and Active [12] SDSNC, proposed for single objective optimization problems. Both strategies increase the number of subpopulations when all subpopulations are stuck. The difference between them is that Classic SCSNC removes the subpopulation when there is another one investigating the same or a similar part of solution space size. Active SDSNC removes all subpopulations except the one containing the global best when the globally best-found solution is improved. In this paper, both these strategies are extended to multi-objective optimisation and used for real-world problems.

[2]https://www.docker.com/

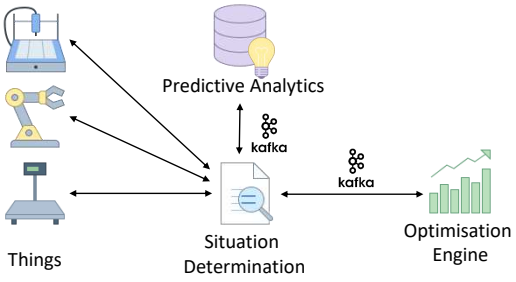

Figure 1: Overall architecture of the proposed system

Leclerc et al. [9] proposed a cloud-based framework facilitating large scale evolutionary experiments. Their framework provided a master-slave architecture with nodes communicating via JSON over HTTP. In the proposed approach, JSON is also used for configuration serialisation, but it is transmitted over the reliable streaming platform named *Kafka*. Kafka has been also used in a simple proof-of-concept genetic algorithm implementation in [6], but no implementation details nor experimental results have been provided there. Also, both these works have not considered a dynamic number of slave nodes.

The framework presented in [3] has executed a genetic algorithm on several machines in accordance with the *serverless* computing paradigm. Such an approach can be considered *pure functional* in that there is no state to maintain between invocations and consequently no server is required to be instantiated in advance. Consequently, the available computing resource is practically unbounded and there is no need for keeping any computation on hold but the payment is made only for the real computation time of the computing resources. In this paper, a similar deployment scheme has been applied, but rather than perform an earlier termination of the optimisation process, we dynamically scale up the number of containers performing the optimisation in parallel to obtain better results in shorter time.

3 SYSTEM ARCHITECTURE AND PROBLEM DESCRIPTION

The optimisation problems considered in this paper are related to an integrated process planning and scheduling in Industry 4.0, where smart plants are expected to manufacture small batches/series of assorted commodities just after receiving a manufacturing order. In the proposed solution,

the optimisation is performed by Optimisation Engine (OE), which is a module of a larger system whose overall architecture is sketched in Figure 1. The optimisation process is triggered by a smart plant's state change, that may be caused by either obtaining a new manufacturing order or by detecting some unexpected plant condition. The latter is possible by connecting plant devices (things) to the Situation Determination (SD) module. This module applies a specific use-case situation model based on a common situation model specified by an ontology to find out the current situation of products, machines and/or processes. The SD module extracts relevant information from the raw data fetched from the things and forwards them to the Predictive Analytics (PA) module that applies machine-learning-based techniques to find certain patterns in that data and predict future parameters of the manufacturing process.

When SD detects any relevant plant state change, it triggers the OE module to perform the plant reconfiguration. The current state is transmitted to OE using a popular distributed streaming platform named Kafka[3] in the form of a message following a well-defined textual protocol. This message includes three types of values, each being either numeric (real or integer) or enumerated: key objectives that shall be optimised, controlled variables that can be mutated during the search-based optimisation process and observable values that carry important information about the plant state, for example, unavailability of a certain plant resource. After receiving such a message, OE performs an optimisation process using an objective function customised for a given plant. In the considered cases, the objective function is based on the interval algebra described in [4]. In the proposed solution, this function is generated automatically based on a factory description specified by an XML-based format, derived from the common situation model. Then, the best-found set of the control variables is returned to SD for applying to the plant.

The functionality of the SD and PA modules, as well as the appropriate objective function creation from a plant description, are crucial for the successful plant reconfiguration. However, all these topics are out of the scope of this paper, which focuses on the functionality and deployment of OE.

4 CLOUD DEPLOYMENT

One of the most popular techniques facilitating cloud computation is Docker. Docker creates and executes containers, i.e., software units that package up a code and all its dependencies. The Docker technology separates application dependencies from infrastructure and hence containers can be executed on any computer executing Docker Engine. Currently, Docker Engine can be run locally under Linux and Windows OS, as well as in data centres and clouds. Several computers with Docker Engine can be joined in a single cluster with container-orchestration systems, among which Kubernetes[4] is the most popular. Kubernetes clusters are available in all major cloud facilities, including AWS, Azure, CloudStack, GCE, OpenStack, OVirt, Photon, IBM Cloud Kubernetes Service, as well as can be installed locally.

[3] https://kafka.apache.org/

[4] https://kubernetes.io/

Due to the convenience of having just a single software unit executable on virtually any machine, OE has been delivered in a form of a Docker container. The optimisation process performed by OE starts after receiving a Kafka message enumerating the key objectives, control and observable metrics and their values. After optimisation, the best found control metrics assignments are sent back to the SD module, using the Kafka streaming platform again. Between two consecutive invocations, OE does not store any information regarding prior optimisation processes and hence is *stateless*. Stateless containers can be executed like functions, in line with the Function as a Service (FaaS) category of cloud computing services, also known as serverless execution. There exists a framework for serverless execution in a Kubernetes cluster named Fission[5]. Fission is capable to auto-scale the number of OE containers in a Kubernetes cluster based on these containers' workload. Correspondingly, the workload depends on the number of islands maintained by an OE container. As this number is decided dynamically using the strategies described in the following section, it is expected that the number of OE containers will automatically scale up and down. During the creation of a new island, the auto-scaler in Fission decides whether that island will be maintained by a new OE container or in one of the already existing OE containers. In the latter case, the load-balancer in Fission will select the container with the lowest load.

5 DETERMINATION OF THE NUMBER OF ISLANDS

In this section, three strategies for island creation and deletion are presented. The number of islands is decided by a master called `manager`. A manager calls several slaves to evolve subpopulations during a certain number of generations, following the popular Island Model of GAs. Two managers, `ManagerClassic` and `ManagerActive`, decide dynamically on the number of islands based on the current optimisation state, whereas the third manager, `ManagerStatic`, uses a fixed, predefined number of islands and acts as the baseline method. The idea behind the dynamic managers has been inspired by their single-objective counterparts from [8, 12].

5.1 `ManagerStatic` Strategy

After obtaining an optimisation request, `ManagerStatic` creates a predefined number, N, of islands. These islands evolve subpopulations using any multi-objective GA, starting from a randomly initiated population including P individuals. During each execution, an island evolves I populations. The manager maintains a Pareto Front approximation, PF, including non-dominated[6] solutions evolved by the islands. Each island is executed S times. As a single execution is referred to as stage, the S parameter is called stage parameter. PF is updated after each stage and, after S stages, is returned as the final result of the optimisation process. This algorithm is outlined in Algorithm 1.

[5] https://fission.io/

[6] A solution in a Pareto Front approximation is non-dominated if it includes the best value among all the solutions in that front for at least one objective.

Algorithm 1: Algorithm of Manager Static

```
inputs  : N: initial number of islands;
          N_max: maximum number of islands allowed;
          S: number of stages;
          I: number of iterations per stage;
          P: number of individuals per island;
outputs : PF: a global Pareto Front approximation maintained by
          the manager;
1  PF = ∅, s = 0;
2  Create N islands with P randomly generated individuals;
3  for s=1,...,S do
4  |  Execute all islands for I iterations;
5  |  Add non-dominated solutions returned from all islands
   |   into PF;
6  |  Make migrations;
   end
7  return PF;
```

After each stage, an inter-island migration is performed in order to exchange partial optimisation results. The migration is performed in the following way. For each island, another island is selected randomly and the best individual from that island replaces the individual with the worst quality. As there are no constraints imposed on the selection of the migration source and its destination, the applied island topology is a fully connected graph. The quality indicator for individuals is arbitrary (e.g., weighted sum with normalisation) and can be customised for the considered optimisation problem.

5.2 `ManagerClassic` Strategy

In contrast to `ManagerStatic`, the island management strategy adopted by `ManagerClassic` alters the number of islands dynamically considering the current state of the optimisation process in the way outlined in Algorithm 2.

Algorithm 2: Algorithm of Manager Classic

```
1  PF = ∅, s = 0;
2  Create N islands with P randomly generated individuals;
3  for s=1,...,S do
4  |  Execute all islands for I iterations;
5  |  Add non-dominated solutions returned from all islands
   |   into PF;
6  |  Make migrations;
7  |  if CI of PF obtained after stage s is higher than that of
   |   stage (s-1) then
   |  |  continue;
   |  else
8  |  |  Delete islands that meet island deletion criteria;
9  |  |  Create one island with P randomly generated
   |  |   individuals;
   |  end
   end
10 return PF;
```

After each stage, the quality of the global Pareto Front approximation, PF, is compared against PF obtained after the previous stage. The fronts are compared using an arbitrary comparator indicator (CI). Without loss of generality, a higher value indicated by the applied CI is assumed to denote a higher front quality. Hence, if the current PF is characterised with a higher CI than the global Pareto Front obtained after the previous step, a solution with a higher quality has been evolved by at least one island. In such a situation, the algorithm continues with the subsequent stage. Otherwise, in Line 8, the manager iterates through the islands sequentially and removes all the islands for which at least one condition from the ones given below is satisfied:

- all individuals have the same genotype,
- another island maintains an identical population,
- the population of another island strictly dominates the population of the considered island[7].

The fulfilment of these conditions indicates that the island population is not likely to contribute to the final result. Hence, it may be more beneficial to create a new island with a random population from scratch (Line 9).

Since `ManagerClassic` monitors periodically the individual quality in each island which, in turn, influences the lifetime of the islands, it may be expected that the quality of the solutions found using this manager will outperform the ones obtained with `ManagerStatic`. This intuition will be confirmed in the experiments described later in this paper.

5.3 `ManagerActive` Strategy

Similarly to `ManagerClassic`, `ManagerActive` also manages the number of islands dynamically. However, the island creation and deletion conditions are different.

Algorithm 3: Algorithm of Manager Active

```
1  PF = ∅, s = 0;
2  Create N islands with P randomly generated individuals;
3  for s=1,...,S do
4  |  Execute all islands for I iterations;
5  |  Add non-dominated solutions returned from all islands to
   |   PF;
6  |  Make migrations;
7  |  if CI of PF obtained after stage s is higher than that of
   |   stage (s-1) then
8  |  |  Delete all islands that do not provide new solutions to
   |  |   PF;
   |  else
9  |  |  Create one island with P randomly generated
   |  |   individuals;
   |  end
   end
10 return PF;
```

As shown in Algorithm 3, after each stage, `ManagerActive` compares the current PF with the one obtained after the previous stage using a CI (Line 7). In case the PF quality has improved, the manager browses through the islands and removes each island which has not evolved any new non-dominated solution during that stage (Line 8). When no

[7] A Pareto Front approximation strictly dominates another if the former contains at least one solution that has a better value than any solutions in the latter for all objectives.

island provides a new non-dominated solution, a new island is created in order to boost the search space exploration (Line 9).

When `ManagerActive` is applied, the lifespan of the island is usually shorter than the ones with the remaining strategies since after each stage a certain number of islands is removed or a new island is created. When a locally optimal solution has been found and the PF quality has not improved for a number of stages, `ManagerActive` maintains more individuals than the strategies described earlier. Consequently, it is more probable to find a better solution in such a situation. On the other hand, if the quality of the global PF improves after each consecutive stage, `ManagerActive` keeps removing islands after each stage regardless of whether the island reached its local optimum or not. This way even the islands likely to influence the final solution can be deleted. The experiments in Section 6 demonstrate the advantages and limitations of this strategy when applied to real-world optimisation problems.

6 REAL-WORLD MANUFACTURING PROBLEMS

In this section, a number of optimisation problems originating from two real-world smart factory use cases is briefly described and used for evaluating the strategies proposed earlier in this paper. As written in the previous sections, the proposed method is agnostic regarding the applied GA or Pareto Front approximation comparator. In all the experiments, a popular multi-objective GA named MOEA/D [16] has been applied and the front qualities are evaluated with the Diversity Comparator Indicator (DCI) [10]. The individuals with the lowest "makespan" (i.e., the manufacturing time) in a subpopulation maintained by a particular island are selected as the migrants. All strategies are executed with the same parameter set, namely $N = 5$, $N_{max} = 10$, $S = 40$, $P = 50$ and $I = 20$. These values have been chosen after many experiments based on a set of popular optimisation problems unrelated to Industry 4.0.

6.1 Process Manufacturing Optimisation Problem

The first scenario is based on a real-world smart factory representing the process manufacturing branch of industry. The considered factory produces different kinds of paint by mixing certain raw materials, in accordance with the predefined set of recipes. The main objective is to decrease the total manufacturing time (makespan) of a given manufacturing order describing the amount of different paint to be produced during a certain day. In the factory, there are 9 mixers (M_1-M_9). Each paint type can be produced using any mixer, but manufacturing time and the amount of paint produced during a single manufacturing process depend on the mixer selection, as shown in Table 1 (the paint type names are in German). Mixers M_1-M_5 are smaller than $M_6 - M_9$. Among the larger mixers, M_8 and M_9 are faster. It is not possible to fill any mixer partially with the raw materials. So, in order to produce an ordered amount of commodities, a multisubset (i.e. a combination with repetitions) of the recipes needs to be selected, allocated to the compatible resources and scheduled in time. As the amount of manufactured paint must be a sum of multiples of the amounts produced by the appropriate resources, a certain surplus of a manufactured paint type may be produced. Storage of such extra paint amount is expensive and hence the surplus of each paint type shall be minimal.

The considered optimisation problem is then characterised with multi-objective criteria, as not only the makespan needs to be minimised, but also the paint surpluses need to be minimal.

If two recipes allocated to the same mixer and executed subsequently manufacture different paint types, a short sequence-dependent setup interval of the length provided by the business partner is inserted between them. This interval models the necessary cleaning process time.

The fitness function for the described plant has been generated automatically from the factory description in the XML format and injected into OE. OE has been Dockerised and deployed to a Kubernetes cluster in the way described earlier in this paper. The amounts of ordered paint equal 45, 40, 30, 20 (in tonnes) of “Std Weiss", “Weiss Matt", “Super Weiss" and “Weiss Basis", respectively. Three managers from Section 5 have been employed to determine the number of islands during the optimisation process. The obtained results are shown in Figure 2. Table 2 details the number of islands' creations, deletions and executions during the entire optimisation process.

Table 1: Process manufacturing example recipes

Paint	Recipe index	Compatible mixers	Amount of produced commodity	Recipe execution time
Std Weiss	1	$\{M_1, M_2, M_3, M_4, M_5\}$	5 t	90 min.
	2	$\{M_6, M_7\}$	10 t	60 min.
	3	$\{M_8, M_9\}$	10 t	45 min.
	4	$\{M_8, M_9\}$	10 t	45 min.
Weiss Matt	5	$\{M_1, M_2, M_3, M_4, M_5\}$	5 t	90 min.
	6	$\{M_6, M_7\}$	10 t	60 min.
	7	$\{M_8, M_9\}$	10 t	45 min.
	8	$\{M_8, M_9\}$	10 t	45 min.
Super Weiss	9	$\{M_1, M_2, M_3, M_4, M_5\}$	4 t	120 min.
	10	$\{M_6, M_7\}$	8 t	90 min.
	11	$\{M_8, M_9\}$	8 t	60 min.
	12	$\{M_8, M_9\}$	8 t	60 min.
Weiss Basis	13	$\{M_1, M_2, M_3, M_4, M_5\}$	6 t	60 min.
	14	$\{M_6, M_7\}$	12 t	45 min.
	15	$\{M_8, M_9\}$	12 t	30 min.
	16	$\{M_8, M_9\}$	12 t	30 min.

Table 2: Dynamic islands changes during one optimisation process

Manager	Island Executions	Islands Created	Islands Deleted
Static	200	5	0
Active	192	29	19
Classic	137	25	23

As stated earlier, `ManagerStatic` maintains the same number of islands (here: $N = 5$) during each from $S = 40$ optimisation stages. `ManagerActive` removes islands not providing

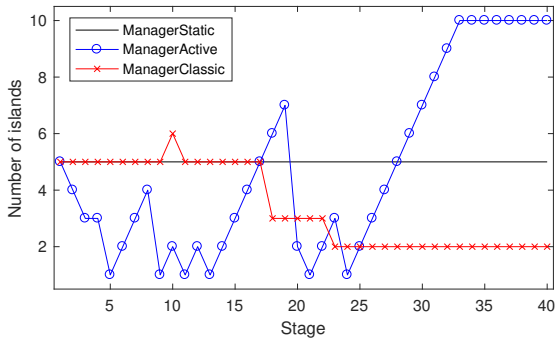

Figure 2: Number of islands during example execution of optimisation using different managers

new globally non-dominated solutions when at least one island has evolved such a new solution. This situation has occurred in the 9th and 20th stages. This manager also creates new islands with random individuals when none provides new globally non-dominated solutions, which is observed between the 13th and 19th stages. `ManagerActive` converged at the 24th stage. In the remaining stages, it adds new islands and finally reaches $N_{max} = 10$, the maximally allowed number of islands, after the 33rd stage.

Initially, `ManagerClassic` has maintained the same number of islands in the majority of stages before reaching the 19th stage. During these stages, each island has provided new non-dominated solutions. After the 19th stage, the number of islands started to decrease as a local optimum has been reached and the manager removes islands that are less likely to contribute to the final solution. At that time, new islands with random individuals are created. This optimisation process converges at the 23rd stage. After the convergence, the manager removes one existing island and adds a new island with random individuals, as no island improves the quality of Pareto Front approximation. `ManagerClassic` executes the lowest number of islands (137 in 40 stages) in comparison with the remaining two strategies. It is characterised also with a higher number of island deletions than `ManagerActive`.

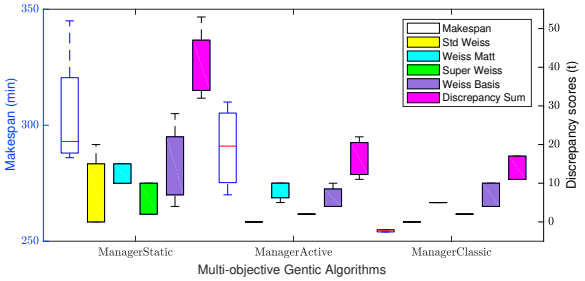

Figure 3: Process manufacturing optimisation results by all managers

The results obtained for the considered optimisation problem are presented in Figure 3. The primary (blue) vertical axis is associated with makespans, which are presented as boxes with a blue frame. Commodity surpluses are labelled as "discrepancy scores" and associated with the secondary (black) vertical axis and presented as boxes with black frames. The magenta boxes show the sum of surpluses of all commodities for each solution and are also associated with the secondary vertical axis. As shown in the figure, `ManagerStatic` is outperformed by both dynamic stategies as `ManagerActive` and `ManagerClassic` have yielded lower values for all objectives. The results of `ManagerActive` are outperformed by `ManagerClassic`, which has also executed significantly fewer island.

The Diversity Comparison Indicator (DCI) [10], a quality indicator commonly applied for assessing the diversity of Pareto Front approximations in many-objective optimisation, has been applied to Pareto Fronts approximations obtained by those three algorithms respectively and returned $\big(0, 0, 1\big)^8$, which indicates that PF returned by `ManagerClassic` strictly dominates PF obtained by either `ManagerActive` or `ManagerStatic`.

As described in Section 5, the rationale behind this observation is that in `ManagerClassic`, an island is removed only if it is less likely to contribute to the final result (e.g., converged completely or being strictly dominated). Instead, `ManagerActive` deletes an island as long as it is more unlikely to contribute to the final solution than the remaining ones. Thus, in general, under identical parameter settings, `ManagerActive` can benefit from a larger number of individuals (due to a higher number of islands) while each island under `ManagerClassic` is usually run for a higher number of stages, which is more favourable in the given optimisation problem.

As the optimisation is performed in a cloud, computational time and the number of islands influence the monetary cost. For each optimisation process described above, the price has been lower than 20$ using Amazon Elastic Container Service for Kubernetes (Amazon EKS) run on 4-cores instances m5 in the AWS London zone. Due to the lowest number of island execution, the optimisation process employing `ManagerClassic` has been the cheapest, whereas maintaining a fixed number of islands in `ManagerStatic` has been the most expensive (11$ for `ManagerClassic` against 15$ for `ManagerStatic` and 12.1$ for `ManagerActive` on 4-cores instances m5.×large).

6.2 Discrete Manufacturing Optimisation Problem

The second real-world use case is related to the discrete manufacturing process of Wire-cut Electrical Discharge Machining (WEDM). WEDM is used for shaping parts made of hard metals by removing unwanted material by using a series of sparks. One of the electrodes is a wire that is constantly wound between two spools. As the wire can be used only once, it is the most expensive consumable in the process.

[8] In general, each numerical value in a tuple obtained with DCI corresponds to a certain front quality in relation to the remaining fronts under comparison. These values are upper-bounded with 1 and a higher value denotes a better relative front quality.

Table 3: Discrete manufacturing example order

Part	Machine size	Manufacturing Way	Cutting time (min)	Wire cost per part ($)	Machine cost per part ($)	Total cost per part ($)
P1	Small	1	2833.5	28.1	164.0	192.1
	Small	2	2956.2	28.1	140.3	168.4
	Small	3	3042.1	28.1	147.8	175.9
	Small	4	3174.1	30.2	136.8	167.0
	Medium	1	2033.5	30.2	242.9	273.1
	Medium	2	2156.2	30.2	208.4	238.6
	Medium	3	2242.1	41.0	196.1	237.1
	Medium	4	2674.1	41.0	189.0	230.0
	Large	1	1256.2	41.0	555.9	596.9
	Large	2	1633.5	53.7	465.6	519.3
	Large	3	1842.1	53.7	427.9	481.6
	Large	4	1974.1	53.7	408.4	462.1

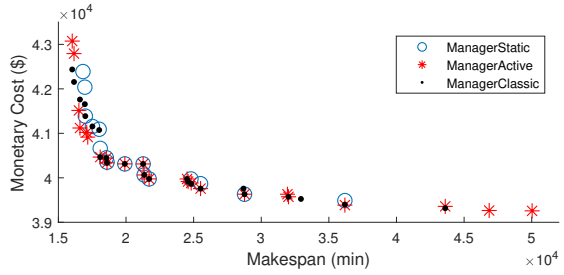

Figure 4: Pareto Front approximations for the considered discrete manufacturing scenario

An example plant is equipped with three WEDM machines of different sizes: “small", “medium" and “large". The larger is the machine, the more expensive its usage. In the considered scenario, 16 metal parts (P1-P16) of different sizes have been ordered. The part sizes are labelled with the smallest machine that can manufacture it, so a “small" part can be manufactured on any machine whereas “large" parts require the large machine. All parts can be manufactured in one of four manufacturing ways (MW) that differ in the types of wire and the machine mode (“eco" or “standard"). An example of 12 various manufacturing configurations for a single part is presented in Table 3.

The goal of the considered optimisation problem is to select a machine to manufacture each part, decide on its MW and schedule the production in time so that the total manufacturing cost and the overall makespan is minimised. In contrast to the previous use case, objectives are explicitly contradictory since a shorter makespan is obtained on a larger machine and hence it incurs a higher monetary cost.

Figure 4 presents the optimisation results obtained with the proposed algorithms using the following parameters: $N = 5$, $N_{max} = 10$, $S = 40$, $P = 50$ and $I = 20$. As shown in this figure, each proposed strategy yields a set of solutions that are evenly distributed on the Pareto Front approximation. The approximation generated by `ManagerStatic` is characterised with lower ranges for both the objectives than the approximations generated with the dynamic strategies. `ManagerActive` has yielded a Pareto Front approximation with the largest ranges with respect to both the objectives. This observation has been confirmed with the DCI test, which returned values $\left(0.852, 1.0, 0.926\right)$ for `ManagerStatic`, `ManagerActive` and `ManagerClassic`, respectively. It can be then concluded that, in contrast to the previous use case, where the objectives could be minimised at the same time and `ManagerClassic` performed the best, in this use case `ManagerActive` delivered results with the highest quality.

The changes in the number of islands during the optimisation process are similar to the trends shown in Figure 2. In case of `ManagerActive`, the islands are again created or removed frequently, reaching the maximal possible number $N_{max} = 10$ at the end of the optimisation. With `ManagerClassic`, the island number remains stable initially and decreases gradually at later stages. The objective function is much simpler than in the previous use case. Hence, the total computation cost for performing this optimisation is significantly lower: it is below 0.5$ regardless of the applied manager when using the same cloud instances as for the considered process manufacturing problem.

In order to evaluate the characteristics of the proposed optimisation scheme when applied to larger problems, the original number of machines, the number of instances of ordered parts and the number of MWs have been multiplied by scale factor $i = 1, \ldots, 10$. For example value $i = 2$, each part will be produced twice and a table corresponding to Table 3 would have 24 rows with possible selections of machine sizes and MWs. The cutting time and costs have not been scaled. The three managers have been used to optimise 30 randomly generated manufacturing orders. In the majority of cases (28 out of 30), `ManagerActive` has performed the best. `ManagerClassic` has acted better than `ManagerStatic` in 27 cases. It may be then concluded, that using `ActiveManager` is the most effective for the considered problem. The dynamic strategies significantly outperform `ManagerStatic`. These conclusions were confirmed by Sign Test with the probability exceeding 99%.

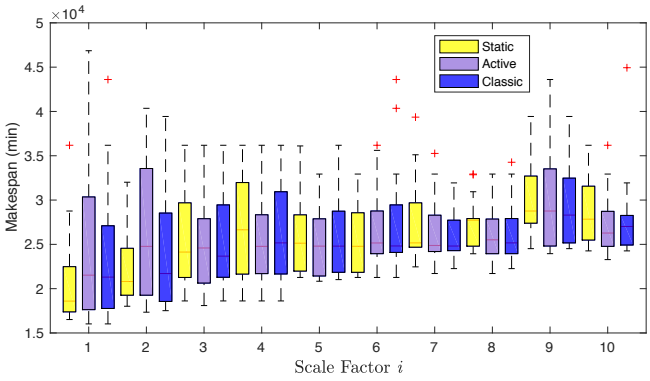

Figure 5: Makespan optimisation results of the proposed cloud-based approaches by scaling the problem size

The box plot presenting the optimisation results for makespan for all the scaled scenarios are presented in Figure 5. The results follow a slowly increasing trend, which implies that the majority of the parts are produced in parallel, benefiting from

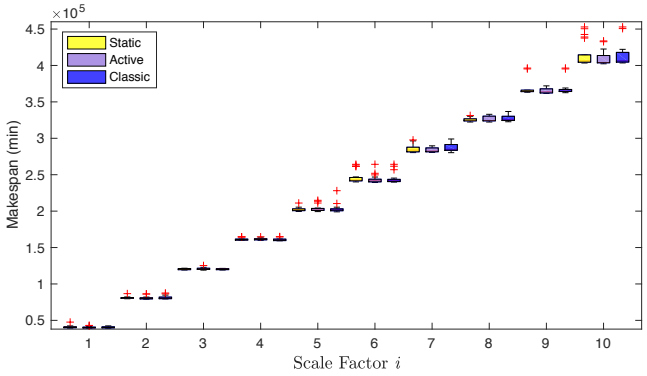

Figure 6: Monetary cost optimisation results of the proposed cloud-based approaches by scaling the problem size

Table 4: DCI of proposed algorithms on scaled discrete manufacturing factory.

i	ManagerStatic	ManagerActive	ManagerClassic
1	0.2	0.747	0.72
2	0.125	0.583	0.417
3	0.217	0.609	0.435
4	0.158	0.842	0.158
5	0.00	0.556	0.444
6	0.043	0.609	0.435
7	0.056	0.889	0.152
8	0.083	0.792	0.208
9	0.00	0.889	0.111
10	0.00	0.952	0.048

the higher number of machines. As expected, `ManagerStatic` is outperformed by the dynamic managers in all cases. In general, `ManagerActive` has delivered Pareto Front approximations with a higher diversity than `ManagerClassic`, which is easily visible in the figure for $i \geq 7$.

In contrast to the makespan, objective "monetary cost" for the scaled scenarios follows an observable linearly increasing trend, which is visible in Figure 6. This is inevitable as the manufacturing of each part is incurred with a certain cost due to the machine activity and wire usage. For this objective, the optimisation results also remain predictable and understandable while scaling the problem size. Table 4 presents the results from the DCI test that again confirms that `ManagerActive` is more favourable for this optimisation problem.

The objective function is computationally less demanding than in the previous use case. Consequently, even for the largest considered factory (i.e., for $i = 10$) the cost of cloud optimisation is low. To maintain the islands, several EC2 instances in the AWS cloud have been used whose ECU[9] ranged from 13 to 68. In average, an execution of a single stage has lasted 907s and the optimisation has cost less than 10$ on all the considered EC2 instance types.

[9] 1 ECU is defined by Amazon as the compute power of a 1.0-1.2GHz of a server CPU from 2007.

In summary, this section demonstrates the efficiency of the proposed cloud-based optimisation algorithms via two integrated process planning and scheduling problems for two real-world smart factories. We also demonstrated that the proposed approach is practically feasible with the acceptable cost incurred for using cloud computing services. From the results, `ManagerClassic` is more favourable where objectives do not conflict with each other (i.e., can be minimised at the same time) while `ManagerActive` benefits optimisation problems that require more individuals during each stage to list more possible solutions for conflicting objectives.

7 CONCLUSION AND FUTURE WORK

In this paper, two genetic algorithms for multi-objective optimisation using a dynamic number of islands have been proposed. The software implementation of these algorithms has been deployed to a cloud and applied to an integrated process planning and scheduling for two real-world smart factories representing the process and discrete manufacturing branches. The presented experimental results have confirmed the superiority of the proposed method over the typical approach using a static number of islands in terms of solution quality and computation time.

In our future work, we plan to investigate other migration topologies, e.g. a ring, different from the fully connected graph used in this paper. Other migration strategies, such as increasing the number of migrants or the strategy of migrants' selection and individuals' replacement will be also evaluated. The innate barrier imposed after each stage is planned to be lifted by proposing a fully distributed algorithm for the subpopulation evolution in the islands. Finally, customised islands management model and GA operators will be proposed to further improve the optimisation results, similarly as in [5].

ACKNOWLEDGMENT

The authors acknowledge the support of the EU H2020 SAFIRE project (Ref. 723634).

The icons used in this paper have been created by Icons8, https://icons8.com.

REFERENCES

[1] Yazen Alsafi and Valeriy Vyatkin. 2010. Ontology-based reconfiguration agent for intelligent mechatronic systems in flexible manufacturing. *Robotics and Computer-Integrated Manufacturing* 26, 4 (2010), 381 – 391. https://doi.org/10.1016/j.rcim.2009.12.001

[2] S. Di Martino, F. Ferrucci, V. Maggio, and F. Sarro. 2012. *Towards migrating genetic algorithms for test data generation to the cloud.* 113–135 pages. https://doi.org/10.4018/978-1-4666-2536-5.ch006

[3] Piotr Dziurzanski, Jerry Swan, and Leandro Soares Indrusiak. 2018. Value-based Manufacturing Optimisation in Serverless Clouds for Industry 4.0. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '18)*. ACM, New York, NY, USA, 1222–1229. https://doi.org/10.1145/3205455.3205501

[4] Piotr Dziurzanski, Jerry Swan, Leandro Soares Indrusiak, and Jose Ramos. 2019. Implementing Digital Twins of Smart Factories with Interval Algebra. In *IEEE International Conference on Industrial Technology*.

[5] Piotr Dziurzanski, Shuai Zhao, Jerry Swan, Leandro Indrusiak, Sebastian Scholze, and Karl Krone. 2019. Solving the Multi-Objective Flexible Job-Shop Scheduling Problem with Alternative Recipes for a Chemical Production Process. In *Applications of*

Evolutionary Computation - 22nd International Conference (EvoApplications 2019).

[6] José-Mario García-Valdez and Juan-Julián Merelo-Guervós. 2018. A Modern, Event-based Architecture for Distributed Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '18)*. ACM, New York, NY, USA, 233–234. https://doi.org/10.1145/3205651.3205719

[7] O. Givehchi, K. Landsdorf, P. Simoens, and A. W. Colombo. 2017. Interoperability for Industrial Cyber-Physical Systems: An Approach for Legacy Systems. *IEEE Transactions on Industrial Informatics* 13, 6 (Dec 2017), 3370–3378. https://doi.org/10.1109/TII.2017.2740434

[8] Halina Kwasnicka and Michal Przewozniczek. 2011. Multi Population Pattern Searching Algorithm: A New Evolutionary Method Based on the Idea of Messy Genetic Algorithm. *IEEE Trans. Evolutionary Computation* 15 (2011), 715–734.

[9] Guillaume Leclerc, Joshua E Auerbach, Giovanni Iacca, and Dario Floreano. 2016. The seamless peer and cloud evolution framework. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM, 821–828.

[10] M. Li, S. Yang, and X. Liu. 2014. Diversity Comparison of Pareto Front Approximations in Many-Objective Optimization. *IEEE Trans. on Cybernetics* 44, 12 (Dec 2014), 2568–2584.

[11] Ning Ma, Xiao-Fang Liu, Zhi-Hui Zhan, Jing-Hui Zhong, and Jun Zhang. 2017. Load balance aware distributed differential evolution for computationally expensive optimization problems. In *GECCO Proceedings Companion, 2017*. ACM, 209–210.

[12] Michal Przewozniczek. 2016. Active Multi-Population Pattern Searching Algorithm for Flow Optimization in Computer Networks - The Novel Coevolution Schema Combined with Linkage Learning. *Inf. Sci.* 355, C (Aug 2016), 15–36.

[13] Pasquale Salza, Filomena Ferrucci, and Federica Sarro. 2016. Develop, Deploy and Execute Parallel Genetic Algorithms in the Cloud. In *GECCO Proceedings Companion, 2016*. ACM, 121–122.

[14] J. Wan, S. Tang, D. Li, M. Imran, C. Zhang, C. Liu, and Z. Pang. 2019. Reconfigurable Smart Factory for Drug Packing in Healthcare Industry 4.0. *IEEE Transactions on Industrial Informatics* 15, 1 (Jan 2019), 507–516. https://doi.org/10.1109/TII.2018.2843811

[15] Shiyong Wang, Jiafu Wan, Daqiang Zhang, Di Li, and Chunhua Zhang. 2016. Towards smart factory for industry 4.0: a self-organized multi-agent system with big data based feedback and coordination. *Computer Networks* 101 (2016), 158 – 168. https://doi.org/10.1016/j.comnet.2015.12.017 Industrial Technologies and Applications for the Internet of Things.

[16] Q. Zhang and H. Li. 2007. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Trans. on Evolutionary Computation* 11, 6 (Dec 2007), 712–731.