Cloud-based Integrated Process Planning and Scheduling Optimisation via Asynchronous Islands

Shuai Zhao[†], Haitao Mei[‡], Piotr Dziurzanski[†], Michal Przewozniczek[†], and Leandro Soares Indrusiak[†]

[†]Department of Computer Science, Univ. of York, Deramore Lane, Heslington, York, YO10 5GH, UK
[‡] IBM York, The Catalyst, Baird Ln, York, YO10 5GA, UK

Abstract. In this paper, we present Optimisation as a Service (OaaS) for an integrated process planning and scheduling in smart factories based on a distributed multi-criteria genetic algorithm (GA). In contrast to the traditional distributed GA following the island model, the proposed islands are executed asynchronously and exchange solutions at time points depending solely on the optimisation progress at each island. Several solutions' exchange strategies are proposed, implemented in Amazon Elastic Container Service for Kubernetes (Amazon EKS) and evaluated using a real-world manufacturing problem.

Keywords: Optimisation as a Service · Multi-objective Genetic Algorithm · Island Model · Amazon EKS · Integrated process planning and scheduling

1 Introduction

In numerous real-world manufacturing scenarios, optimisation of the manufacturing plan and its scheduling seem to be ideally suited to be conducted in a cloud. Firstly, they are notorious to require substantial computation and secondly, as the optimisation process is triggered when a smart factory state changes, the needs for huge computational resources are interleaved with idle intervals. Yet the problem of cloud-based optimisation of realistic industrial problems is relatively unpopular in academia [1]. One of the reasons is the innate heterogeneity of the manufacturing process, as discussed, e.g., in [2], and hence the difficulties in proposing an optimisation framework generic enough to be applicable to a wide range of manufacturing optimisation problems.

The main ambition of the project summarised in this paper is to propose a cloud-based service capable of optimising real-world manufacturing problems ranging from discrete manufacturing (i.e., production of distinct items) to process manufacturing (i.e., production using formulations or recipes). The knowledge description regarding the smart factory and the manufacturing order to be processed can be specified using a dedicated ontology, for example, based on a common ancestor ontology for generic manufacturing domain proposed in [3].

Such ontology can be then used to build an analytic description of a smart factory (aka digital twin), as proposed in [4], which then can evaluate alternative manufacturing configurations as a part of a search-based optimisation process. As this process is executed in the cloud on-demand only, the applied computing resources can be relatively powerful and large in quantity as long as the computation time is not long-lasting. It is then suitable to perform distributed computation in a cluster with several computing nodes. Among search-based optimisation meta-heuristics, the island model of genetic algorithms (GAs) is applicable to distributed execution. In this model, each node evolves 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 [5]. Although a cloud-based realisation of a traditional island-model is quite effective [6], the underlying synchronous execution of each generation may be treated as a source of potential performance loss due to the risk of island failures, different processing time or communication latency. Removing these drawbacks was our main motivation behind proposing a new, asynchronous version of the island model, where the islands exchange their migrants only through a fast NoSQL database at time points decided by the islands based on the progress of their local optimisation process. A general algorithm is proposed and several migration strategies are implemented, deployed to a Kubernetes cluster (using Amazon EKS) and evaluated based on a real-world scenario, formulated by EU H2020 SAFIRE project partner who was in charge of the evaluation based on a real discrete-manufacturing use case.

The main contribution of this paper can be summarised as follows: (i) proposing a generic algorithm for asynchronous island model with multiple objectives, (ii) suggesting several migration strategies for the proposed asynchronous island model, (iii) proposing a cluster-based architecture following the proposed asynchronous island model in Amazon EKS, (iv) presenting experimental evaluations of the suggested migration strategies based on a real-world manufacturing scenario.

2 Related work

Genetic Algorithms (GAs) have been arguably one of the most widely-used optimisation meta-heuristics since the seminal work of John Holland in 1960s. In GAs, a population of solutions to a particular problem is improved generation after generation, mimicking the breeding of living organisms. The solutions represented by chromosomes are selected with a probability proportional to their 'fitness', crossed over and mutated. Despite the initial population is randomly generated, the subsequent generations are increasingly closer to the optimal solution. The original GAs were executed sequentially and hence they were notorious for low speed [7]. Several techniques have been proposed to alleviate this problem, including parallel execution of GAs [5, 8, 9].

The typical parallelisation of GAs can be performed either at the fitness-evaluation or the population level (the island model), performed synchronously following the master-slave architecture [5]. In clouds, these approaches are beneficial only under certain conditions, since the nodes are heterogeneous and con-

nected with links characterised with different latencies. The fitness-evaluation level parallelism is beneficial only for expensive fitness functions [8], whereas the barrier applied in the island model is detrimental when the slave nodes are unreliable or have assorted response times [9]. In order to find approaches more suitable for contemporary cloud clusters, it is beneficial to undust the research related to evolutionary Peer-to-Peer (P2P) computing, as they assume varied response time and nodes' unreliability. For example, in [10], a number of evolutionary strategies for multi-objective P2P optimisation has been evaluated, such as distributed migration decision criterion, exchange topology, number of emigrants, emigrants selection policy and replacement/integration policy. In this paper, we investigate similar criteria but for a different distributed algorithm, cloud architecture and when applied to a real-world manufacturing problem.

In the proposed solution, a custom multi-objective GA has been containerised using Docker [11], similarly as recommended in positional paper [12]. In contrast to that paper, we deployed the containers in a Kubernetes cluster [13]. The islands communicate each other using a NoSQL database rather than an open-source message broker named RabbitMQ. However, the performance of both the solutions is difficult to compare as the authors of that paper provided no implementation details nor the experimental results. In contrast, this paper describes a series of experiments based on real-world industrial scenarios.

Ma et al. employed the population-level parallelisation in [9]. Their solution followed the master-server architecture. The number of slaves was decided statically. Each slave obtained a subpopulation of the size inversely proportional to its CPU utilisation. Then the corresponding fitness values were computed and returned to the master. In the proposed solution, the number of Kubernetes worker nodes is decided dynamically using the auto-scaling facility provided by Amazon EKS, triggered with an alarm monitoring the memory usage of the nodes.

A simple yet interesting proof-of-concept GA implementation described in [14] applied the island model of GA. The islands have been executed in the serverless manner which leverages the scaling capabilities of that solution. However, no implementation details nor experimental results were provided to back the claims regarding the performance of that proposal. The serverless Function-as-a-Service facilities offered nowadays by popular cloud vendors impose strict limitations on a function execution both in timeout and consumed memory. For example, Amazon Lambda in May 2019 was limiting the maximal invocation payload, consumed memory and deployment package size to 256 KB, 3008 MB, zipped 50 MB, respectively. Hence, it is unclear whether the architecture from [14] can be applied in practice with real-world scenarios as analysed in this paper. One of the possibilities of omitting these limits in serverless execution is to use Fission, a popular framework for serverless functions on Kubernetes, as proposed in [6]. However, that paper still followed a traditional master-slave architecture with an innate barrier at the end of each optimisation stage. In contrast, the solution proposed in this paper is fully distributed and the nodes are executed asynchronously. The number of nodes is decided by the Kubernetes horizontal auto-scaler based on the node utilisation rather than the master node as proposed in that paper.

3 Asynchronous Island-based GA with Migrations

In the island model of GA, the evolution is performed independently on a number of subpopulations by GA instances named "islands". Aperiodically, the islands exchange the migrants. The traditional island model follows a fully synchronous master-slave architecture: the iterations on all islands begin at the same time, triggered by the master node, and the iteration completion is synchronised with a barrier. However, this approach can be modified to be fully distributed. In this section, the asynchronous island-mode GA is depicted in Algorithm 1 with several migration strategies suggested.

Each island in the island mode of GA maintains its own subpopulation. It searches towards the optimal solution within a given number of execution stages, where each execution stage contains a fixed number of iterations. The optimisation engines run in each island are executed asynchronously and do not communicate directly with each other. Instead, they communicate using a light-weight database (see GA Data Service in Section 4), pushing their selected solutions at certain time points. At other time points, the solutions pushed by other islands are popped and applied by an island to modify its current Pareto Front approximation. Similarly to [15], a complete migration is performed by a selection and a replacement operator. The former selects the migrants to be pushed to a database and possibly later imported (popped) by other islands, whereas the latter operator selects the individuals in the Pareto Front approximation in an island that will be replaced by the migrants popped from a database so that the same population size is maintained during the entire execution. In each island, the optimisation process stops after evolving a predefined number of generations.

In this paper, four strategies for implementing the selection operator are considered, as enumerated below:

- *Generic selection* does not perform the actual selection from the current Pareto Front approximation but, instead, it randomly generates a new solution. This strategy serves as the performance baseline for the remaining selection operators.
- *Random selection* randomly selects a solution from the current Pareto Front approximation.
- *Best selection* selects the best solution from the current Pareto Front approximation. The solution quality is evaluated with the Generational Distance (GD) performance indicator from [16], which quantifies the proximity of a given solution to the ideal point.
- *Diversity selection* selects the solution with the highest diversity based on the Crowding Distance (CD) value [17], which measures the average distance between the solution and its two closest neighbours in the current Pareto Front approximation.

To maintain a fixed size of each island's population, a certain replacement operator is required to be applied during the migration. This paper considers two replacement strategies:

- *Random replacement* removes a randomly selected solution in the population of the target island.

Algorithm 1: Asynchronous island-based GA

```
inputs  : I: number of iterations; P: number of individuals per island;
          S: number of stages; R: number of maximum stuck iterations in a row;
          M: number of solutions to migrate; CI: quality indicator;
outputs : PF: a Pareto Front (PF) approximation;
1  PF = ∅, s = 0, c = 0;
2  create a GA island with P randomly generated solutions;
3  for s=1,...,S do
4      execute the GA island for I iterations;
5      add non-dominated solutions returned into PF;
6      if CI value of PF obtained after stage (s) is not higher than that of stage
         (s-1) then
7          increment c;
           if c == R then
8              c=0;
9              push the PF approximation to database;
           end
10         for m=1,...,M do
11             pull a PF approximation from a database;
12             migrate one solution from the remote set to the current population;
           end
       end
   end
13 push the final PF approximation to database;
```

– *Worst replacement* removes the worst solution in terms of the solution quality based on a certain quality indicator.

With the above selection and replacement operators combined, we provide, in total, eight migration strategies that can be pre-configured before the optimisation process.

Algorithm 1 starts with P randomly generated solutions and then executes for S stages, where each stage contains I iterations. After the GA island is executed in each stage, an approximation of Pareto Front, PF is updated with new non-dominated solutions (if there exist any). Then, a quality indicator[1] is applied to check the quality of the current Pareto Front approximation and is compared to that of the approximation in the previous execution stage. If the quality is not improved continuously over the prior R iterations (i.e., stuck in a local optimum), the Pareto Front approximation is *pushed* to the database by overriding the previous approximation set of this island (if it exists). In addition, after each execution stage that does not improve the Pareto Front approximation, a *pull* operation is performed to get solutions from a Pareto Front approximation from other islands, randomly selected, in the database (if there exits any). Then migrations are performed to migrate M solutions from the selected front to the current population based on a certain selection and replacement operators

[1] We do not enforce the choice of quality indicator applied in the algorithm, but assume that a higher quality value indicates a higher quality of the optimisation result.

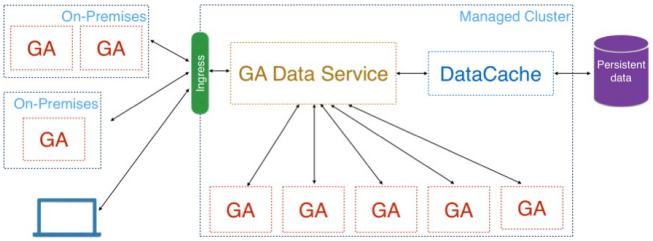

Fig. 1. The architecture of the distributed island-based GA optimisation algorithm.

described previously. Lastly, the PF approximation is pushed to the database as the final optimisation result obtained by this GA island.

4 Cloud Deployment

Section 3 described the GA with asynchronous islands. To deploy this algorithm in a cloud environment, the architecture depicted in Figure 1 is applied. It contains the following components:

- `GA Data Service (data tier)` is responsible for the data communication between islands and storing the data in a persistent data storage.
- `Data Cache` is used to reduce the response time when the data service reads/writes data from/to the persistent storage.
- `GA Island` executes the proposed GA; it can run either on a managed cluster or on-premise.

`GA Data Service` is highly available and automatically scale out/in according to the load of the requests from GA islands. Additionally, the data cache is designed to use a distributed key/value storage, such as a Redis cluster or Cassandra, to support both high availability and fast data exchange. The microservice architecture employed by the proposed solution decouples the components so that the whole solution can be easily deployed to any distributed system. This enables this solution to be provided as a cloud service by cloud providers and requires the minimum possible maintains.

The deployment of the proposed architecture is based on the following assumptions:

- The number of islands that are running at the same time is up to hundreds.
- These islands issue requests to data-tier servers in a sporadic fashion, i.e., the requests (both sending data to or requesting data from the data-tier) arrive with a minimum interval, longer that the data-tier servers' response time.
- The amount of data exchange between the islands and the data-tier is relateively low, up to a few MBs in a single push/pop operation.

In the past several years, Docker [11] and Kubernetes [13] are two popular techniques for containerisation and container orchestration, respectively. Docker allows applications to be shipped to any popular operating systems by creating

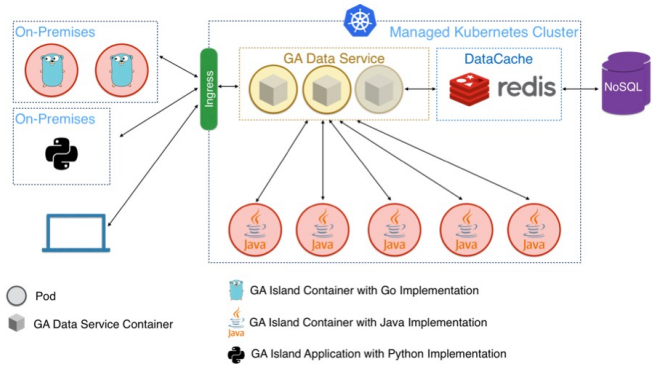

Fig. 2. The architecture of the cloud-base manufacturing Planning and scheduling optimisation system.

a Docker image that is similar to a virtual file system so that the application and its dependencies are encapsulated together. A Docker image is instantiated as a running container by the underlying execution-engine, such as Docker Engine or containterd. Kubernetes is a platform running on a computer cluster, and provide container orchestration functionalities, such as component abstraction (e.g., Pod, Service), DNS service, software-defined network, resource allocation, load balancing etc. Additionally, Kubernetes also provides Horizontal Pod Autoscaler (HPA) to dynamically auto-scale out/in the replicas of a service component based on several metrics, for example, the CPU or memory utilisation. The Cluster Autoscaler (CA) is used to dynamically adjust the number of computing nodes in a Kubernetes cluster. Lastly, Kubernetes allows different plugins to be installed. In this paper, we employ an ingress controller to allow users/applications to communicate with the data-tier service outside of the cluster.

Docker and Kubernetes have been adopted by many providers such as Amazon AWS, Microsoft Azure, Google Cloud, and IBM Cloud. This enables us to leverage the managed Kubernetes services from these cloud providers, rather than installed locally on premises. The Kubernetes HPA and CA enable autoscaling the components (such as the data-tier service) in our system based on the load. By creating multiple instances of service components, Kubernetes automatically handles the load balancing and re-starts a faulty container once detected. This approach enables the proposed system to be highly available during the operation.

Figure 2 depicts the deployment of the system presented in this paper. The core component is `GA Data Service`, which is responsible for data exchange between islands and also generating reports to users. It has a minimum number of instances by default to provide service high availability and scaling out/in according to the load of the requests. The islands can be implemented using any programming language, and communicate with the GA Data Service via REST API from within the cluster or outside of the cluster through the ingress. The GA Data Service stores all the data into an external NoSQL cluster and uses a Redis cluster as a cache layer.

Table 1. Cost of *push* operation with a scaled number of islands (in ms)

No. islands		2	4	8	16	32	64	128
push	avg.	2013.78	2012.92	2013.52	2013.79	2016.33	2014.24	2017.78
	std.	7.61	1.62	1.66	2.20	70.89	31.80	77.69
pull	avg.	1000.56	1000.52	1000.54	1000.64	1002.53	1001.49	1002.56
	std.	0.52	0.51	0.56	0.59	44.79	31.67	44.77

5 Experimental Results

This section investigates the efficacy of the proposed cloud-based deployment of the GA algorithm based on the asynchronous island management model when applied to the real-world use case described in [4], in which a set of 14 metal parts is ordered to be manufactured in a plant equipped with 12 Wire Electrical Discharge Machining (WEDM) machines of 3 different sizes, operating in 3 various modes each. The considered optimisation problem is an example of a typical manufacturing planning and scheduling problem that can be found in any discrete manufacturing company. However, the used optimisation engine can be also applied to process manufacturing as described in [1].

We first evaluate the communication overhead of the proposed cloud architecture by measuring the response time of *push* and *pull* operations. The evaluation is conducted on Amazon Elastic Container Service for Kubernetes (Amazon EKS) run on dual-cores t2.medium instances in the AWS US-West-2 Zone spans over its all availability zones. Table 1 reports the average response time (in milliseconds) and the standard deviation for 10,000 push and pull operations. The size of data for each push and pull operations equals 590KB (i.e., a Pareto Front approximation with 50 elements). According to the results, the overhead incurred due to communication is relatively even despite increasing the number of asynchronous islands, which repeatably issue asynchronous push and pull requests to the data service. As observed, a push operation takes about 2 seconds and a pull operation needs about 1 second to complete regardless the number of islands.

The following experiment investigates the efficiency of the proposed optimisation algorithm (recall Algorithm 1) with eight migration strategies described in Section 3. The optimisation algorithm is configured with $S = 40$, $P = 50$, $I = 20$, $R = 3$, $M = 1$. The Diversity Comparator Indicator (DCI) [18] has been used as the quality indicator CI for measuring the quality of Pareto Front approximations obtained in the subsequent execution stages. DCI quantifies the diversity of the given approximation similarly as in [1]. Five asynchronous islands have been created in the Kubernetes cluster. In total, 100 test cases have been performed for each migration strategy. A ranking has been constructed for Pareto Front approximations obtained by the evaluated migration strategies in a way that each strategy has received the number of points equal to the number of strategies with lower or equal DCI value.

As shown in Figure 3, the strategy without any migrations (i.e., box *None*) has yielded the result with the worst quality among all the tested strategies. This result was expected as in this strategy there is no communication with other islands and hence it cannot obtain the performance boost via import-

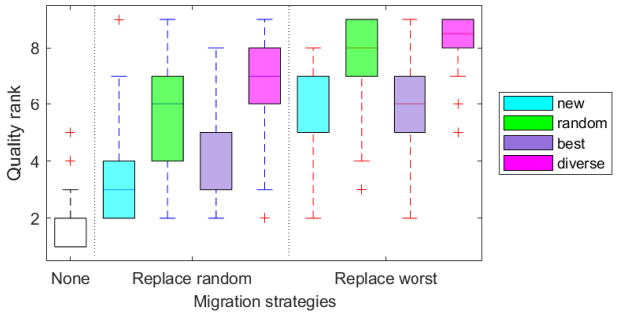

Fig. 3. Optimisation result quality ranking with different migration strategies.

Table 2. Average and median ranking for the considered migration strategies.

Migration strategy	no migration	random + new	random + random	random + best	random + diversity	worst + new	worst + random	worst + best	worst + diversity
avg.	1.596	3.38	5.73	3.75	6.69	5.44	7.82	5.74	8.22
med.	1	3	6	3	7	5	8	6	9

ing non-dominated solutions from other subpopulations (islands). The strategies with the random replacement operator have outperformed the ones applying the worst replacement operator (i.e., *replace worst*) regardless of the selection operator. The *diversity selection* and *best selection* operators have outperformed the remaining selection operators. In accordance with the results presented in [15], migration of the best solution has not been profitable mainly due to the risk of the premature convergence of the entire subpopulation. Instead, migration of the solutions likely to improve the Pareto Front approximation diversity (i.e., *diversity selection* and *random selection* operators) is more beneficial in the studied problem.

Table 2 summarises the average and median ranking values of all the competing strategies. As given in the table, both *random selection* and *diversity selection* operators provide higher quality ranks in average and median values. As expected, the strategy without migration has ranked the lowest. In addition, the *worst replacement* operator has yielded a better average and median quality ranking values than the *random replacement* strategy for each selection strategy. These results are in line with the results presented in Figure 3. In addition, all the above observations have yielded a statistically significant difference according to the Sign Test with p-value threshold for statistical significance equal to 0.02.

Figure 4 reports the numbers of push and pull operations issued by each strategy. The importance of these metrics stems from the cloud communication overhead as shown earlier in Table 1. As shown on the left hand side of the figure, the *no migration*, *new selection* and *best selection* operators have led to a larger number of push operations than the *random selection* and *diversity selection* operators. This observation indicates that the algorithm applying the former

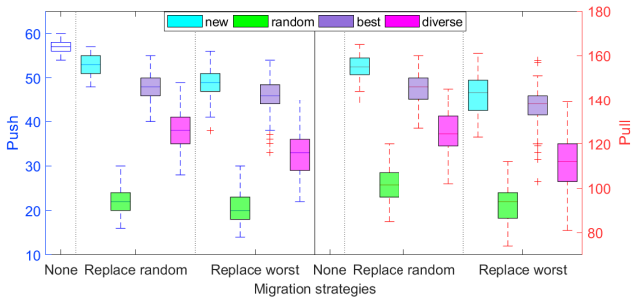

Fig. 4. Push (9 leftmost boxes) and pull (8 rightmost boxes) requests made by each migration method.

Table 3. DCI comparison of the considered migration strategies.

M	None	random + new	random + random	random + best	random + diversity	worst + new	worst + random	worst + best	worst + diversity
1	0.294	0.294	0.294	0.294	0.412	0.294	0.529	0.353	0.706
2	0	0	0.048	0	0.19	0.238	0.381	0.143	0.333
3	0	0	0.143	0	0.238	0.095	0.095	0.048	0.619
4	0	0.042	0.125	0.125	0.25	0.125	0.042	0.042	0.292
5	0	0.08	0.12	0	0.28	0.12	0.12	0.04	0.24

three operators is less likely to escape from local optima since both the push and pull operations are issued when an island has not improved its Pareto Front approximation for a given number of execution stages. In addition, although the *diversity selection* operator has yielded more communication requests than the *random selection* operator, it has usually led to better optimisation results (recall Figure 3). This observation indicates that its relatively heavy communication is beneficial.

Table 3 gives the DCI quality indicator for the strategies with different numbers of solutions transferred during one migration (as specified by parameter M in Algorithm 1). Again, the *diversity selection* operator has led to the best results for all the considered M values and both the replacement operators. In Table 4, DCI values comparing the Pareto Front approximations for different numbers of migrants, M, are presented. From this table, it can be concluded that an increased number of solutions migrated improves the final solution quality. However, having more migrants impose higher communication overheads in the cloud, as discussed earlier in this section.

Standard Amazon EC2 instances have been used as Amazon EKS worker nodes. Their ECUs[2] ranged from 13 to 68. On average, execution of a single stage has taken about 900s and hence the total EC2 cost (including the data transfer cost) has not exceeded 10 USD in any case. Additionally, AWS charged 0.20 USD per hour for using an Amazon EKS cluster in May 2019[3]. These costs

[2] 1 ECU is defined as the compute power of a 1.0-1.2GHz server CPU from 2007.
[3] The current costs can be found at https://aws.amazon.com/eks/pricing/

Table 4. DCI quality changes with an increased number of migrations in one *pull*.

	replace random					replace worst				
M	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
random	0	0.292	0.208	0.292	0.333	0.13	0.174	0.304	0.304	0.348
best	0.043	0.217	0.13	0.478	0.304	0.316	0.158	0.316	0.316	0.526
diversity	0.115	0.154	0.308	0.308	0.192	0.083	0.208	0.292	0.25	0.375

are just 0.02 per cent of the total production cost of the considered parts and hence it is negligible for our business partner.

6 Conclusion

In this paper, a GA for multi-objective optimisation using asynchronous islands have been proposed. The software implementation of these algorithms has been deployed to a Kubernetes cluster (in Amazon EKS) and applied to an integrated process planning and scheduling for a real-world smart factory representing the discrete manufacturing branch. Several migration strategies have been evaluated and the most favourable selection and replacement operators have been identified. Similarly, various numbers of migrants have been analysed.

In our future work, we plan to investigate migration topologies different from the fully connected graph used in this paper, e.g. a ring. A custom scaling of the number of island based on the optimisation state is also planned. Finally, a larger set of real-world manufacturing problems is planned to be evaluated.

Acknowledgement

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

Bibliography

1. Dziurzanski, P., Zhao, S., Swan, J., Indrusiak, L.S., Scholze, S., Krone, K.: Solving the multi-objective flexible job-shop scheduling problem with alternative recipes for a chemical production process. In: Applications of Evolutionary Computation. pp. 33–48. Springer International Publishing (2019)
2. Méndez, C.A., et al.: State-of-the-art review of optimization methods for short-term scheduling of batch processes. Computers & Chemical Engineering 30(6-7), 913–946 (2006)
3. Lemaignan, S., Siadat, A., Dantan, J.., Semenenko, A.: Mason: A proposal for an ontology of manufacturing domain. In: IEEE Workshop on Distributed Intelligent Systems: Collective Intelligence and Its Applications (DIS'06). pp. 195–200 (June 2006)
4. Dziurzanski, P., Swan, J., Indrusiak, L.S., Ramos, J.: Implementing digital twins of smart factories with interval algebra. In: IEEE International Conference on Industrial Technology. ICIT 2019 2019 (2019)

5. Di Martino, S., Ferrucci, F., Maggio, V., Sarro, F.: Towards migrating genetic algorithms for test data generation to the cloud (2012)
6. Zhao, S., Dziurzanski, P., Przewozniczek, M., Komarnicki, M., Indrusiak, L.S.: Cloud-based dynamic distributed optimisation of integrated process planning and scheduling in smart factories. In: Proceedings of the Genetic and Evolutionary Computation Conference. GECCO '19, ACM, New York, NY, USA (2019)
7. Thierens, D.: Scalability problems of simple genetic algorithms. Evol. Comput. 7(4), 331–352 (Dec 1999), http://dx.doi.org/10.1162/evco.1999.7.4.331
8. Leclerc, G., Auerbach, J.E., Iacca, G., Floreano, D.: The seamless peer and cloud evolution framework. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference. pp. 821–828. ACM (2016)
9. Ma, N., Liu, X.F., Zhan, Z.H., Zhong, J.H., Zhang, J.: Load balance aware distributed differential evolution for computationally expensive optimization problems. In: GECCO Proceedings Companion, 2017. pp. 209–210. ACM (2017)
10. Melab, N., Mezmaz, M., Talbi, E..: Parallel hybrid multi-objective island model in peer-to-peer environment. In: 19th IEEE International Parallel and Distributed Processing Symposium. pp. 9 pp.– (April 2005)
11. Enterprise Application Container Platform. https://www.docker.com/, accessed: 2019-04-19
12. Salza, P., Ferrucci, F., Sarro, F.: Develop, deploy and execute parallel genetic algorithms in the cloud. In: GECCO Proceedings Companion, 2016. pp. 121–122. ACM (2016)
13. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/, accessed: 2019-04-19
14. García-Valdez, J.M., Merelo-Guervós, J.J.: A modern, event-based architecture for distributed evolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 233–234. GECCO '18, ACM, New York, NY, USA (2018)
15. Nogueras, R., Cotta, C.: An analysis of migration strategies in island-based multimemetic algorithms. In: International Conference on Parallel Problem Solving from Nature. pp. 731–740. Springer (2014)
16. Ishibuchi, H., Masuda, H., Tanigaki, Y., Nojima, Y.: Modified distance calculation in generational distance and inverted generational distance. In: Gaspar-Cunha, A., Henggeler Antunes, C., Coello, C.C. (eds.) Evolutionary Multi-Criterion Optimization. pp. 110–125. Springer International Publishing, Cham (2015)
17. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In: International conference on parallel problem solving from nature. pp. 849–858. Springer (2000)
18. Li, M., Yang, S., Liu, X.: Diversity comparison of pareto front approximations in many-objective optimization. IEEE Trans. on Cybernetics 44(12), 2568–2584 (Dec 2014)