Performance-Cost Optimization of Moldable Scientific Workflows

Marta Jaros[1][0000−0002−7775−8106] and Jiri Jaros[1][0000−0002−0087−8804]

Brno University of Technology,
Faculty of Information Technology,
Centre of Excellence IT4Innovations,
Brno, Czech Republic
{martajaros,jarosjir}@fit.vutbr.cz

Abstract. Moldable scientific workflows represent a special class of scientific workflows where the tasks are written as distributed programs being able to exploit various amounts of computer resources. However, current cluster job schedulers require the user to specify the amount of resources per task manually. This often leads to suboptimal execution time and related cost of the whole workflow execution since many users have only limited experience and knowledge of the parallel efficiency and scaling. This paper proposes several mechanisms to automatically optimize the execution parameters of moldable workflows using genetic algorithms. The paper introduces a local optimization of workflow tasks, a global optimization of the workflow on systems with on-demand resource allocation, and a global optimization for systems with static resource allocation. Several objectives including the workflow makespan, computational cost and the percentage of idling nodes are investigated together with a trade-off parameter putting stress on one objective or another. The paper also discusses the structure and quality of several evolved workflow schedules and the possible reduction in makespan or cost. Finally, the computational requirements of evolutionary process together with the recommended genetic algorithm settings are investigated. The most complex workflows may be evolved in less than two minutes using the global optimization while in only 14s using the local optimization.

Keywords: task graph scheduling, workflow, genetic algorithm, moldable tasks, makespan estimation

1 Introduction

All fields of science and engineering use computers to reach new findings, while the most compute power demanding problems require High Performance Computing (HPC) or Cloud systems to give answers to their questions. The problems being solved nowadays are often very complex and comprise of a lot of various tasks describing different aspects of the investigated problem and their mutual dependencies. These tasks compose a processing scientific workflow [3]. There are immense of such scientific workflows in various fields [22], yet they have one

thing in common. They all demand to be computed in the minimum possible time, and more often, for the lowest possible cost.

The execution of a scientific workflow on an HPC system is performed via communication with the HPC front-end, also referred to as job scheduler [13]. After the workflow data has been uploaded to the cluster, the workflow tasks are submitted to the computational queues waiting until the system has enough free resources, and all task dependencies have been resolved (predecessor tasks have been finished).

Modern HPC schedulers control multiple processing queues and implement various techniques for efficient task allocation and resource management [15]. However, the workflow queuing time, computation time and related cost are strongly dependent on the execution parameters of particular tasks provided by the user during submission. These parameters usually include the temporal parameters such as requested allocation length, as well as spatial parameters including the number and type of compute nodes, the number of processes and threads, the amount of memory and storage space, and more frequently, the frequency and power cup of various hardware components. These parameters, unfortunately, have to be specified by the end users based on their previous experience with the task implementation and knowledge on the input data nature.

In everyday practice, the estimations of task allocation lengths are quite inaccurate, which disturbs the scheduling process. Most users deliberately overestimate the computational time in order to provide some reserve to mitigate performance fluctuation and prevent premature termination of the task execution [23]. Moreover, many complex tasks are written as moldable distributed parallel programs being able to exploit various amounts and types of computing resources. Nonetheless, it is again the user responsibility to choose appropriate values of these parameters according to the input data.

The task moldability is often limited by many factors, the most important of which being the domain decomposition [6], parallel efficiency [2], and scalability [14]. While the domain decomposition may limit the numbers of processing units (nodes, processes, threads) to rather a sparse list of acceptable values, the parallel efficiency determines the execution time and cost for a given task and a chosen amount of resources. Naturally, the lower the parallel efficiency, the lower the speed-up, and consequently, the longer the computation time and the higher the computational cost. Finally, the scalability upper-bounds the amount of exploitable resources by the overall available memory.

While the field of rigid workflow optimization, where the amount of resources per task cannot be tuned, has been thoroughly studied and is part of common job schedulers such as PBSPro [13] or Slurm [30], the automatic optimization and scheduling of moldable workflows has still been an outstanding problem, although firstly solved two decades ago in [10].

For the last decade, many papers have focused on the estimation of rigid workflow execution time in HPC systems and enhancing the resource management. For example, Chirkin et al. [7] introduces a makespan estimation algorithm that may be integrated into schedulers. Robert et al. [25] gives an overview of

task graph scheduling algorithms. The usage of genetic algorithms addressing the task scheduling problems has also been introduced, e.g., a task graph scheduling on homogeneous processors using genetic algorithm and local search strategies [17] and a performance improvement of the used genetic algorithm [24]. However, handful works have taken into the consideration the moldability and scaling behaviour of particular tasks, their dependencies and the current cluster utilization [9, 4, 29].

This paper focuses on the automation optimization of the moldable scientific workflow execution using genetic algorithms [28]. The optimization of execution parameters is based on collected historical performance data (i.e., strong scaling) for supported tasks in the workflow. The paper presents several objective functions and trade-off coefficients that allow to customize the pressure either on the overall execution time, or the computational cost, or both.

The rest of the paper is structured as follows. Section 2 describes the optimization algorithm, the solution encoding specifying the amount of resources per task, the objective and fitness functions evaluating the quality of the candidate workflow schedule and the details of the applications use cases. Section 3 elaborates on the quality of the genetic algorithm and its best set-up, presents the time complexity of the search process and compares several workflow execution schedules by the optic of particular objective functions. The last section concludes the paper and draws potential future improvements of this technique.

2 Proposed Algorithm

The assignment of optimal amount of compute resources to particular tasks along with the scheduling of the workflow as a whole is known to be an NP-hard problem [9]. There have been several attempts to use heuristics to solve this problem [4, 16, 18, 26], however, they are either tightly connected to an existing HPC cluster and its scheduler, use idealized models of strong scaling and parallel efficiency, or optimize only one criterion such as makespan, cluster throughput, or execution cost. The user tunability of these approaches are thus limited.

Therefore, we decided to use genetic algorithms, which are highly flexible in combinatorial optimization and scheduling [8]. From the vast number of existing implementations, PyGAD [11], an open-source Python library for building the genetic algorithm, was chosen. PyGAD supports various types of genetic operators and selection strategies, and offers a simple interface for objective function definition.

The overall concept of the moldable workflow scheduling optimization using PyGAD is shown in Fig. 1. The structure of the task graph is converted into a 1D array where each element corresponds to a single task and holds its execution parameters. The genetic algorithm traverses the search space and seeks for good solutions by applying genetic manipulations and selection strategies on the population of candidate solutions. The quality of these candidate solutions is evaluated by the fitness function. Although the paper presents three different methods to evaluate the schedule quality, the concept is similar in all cases. First,

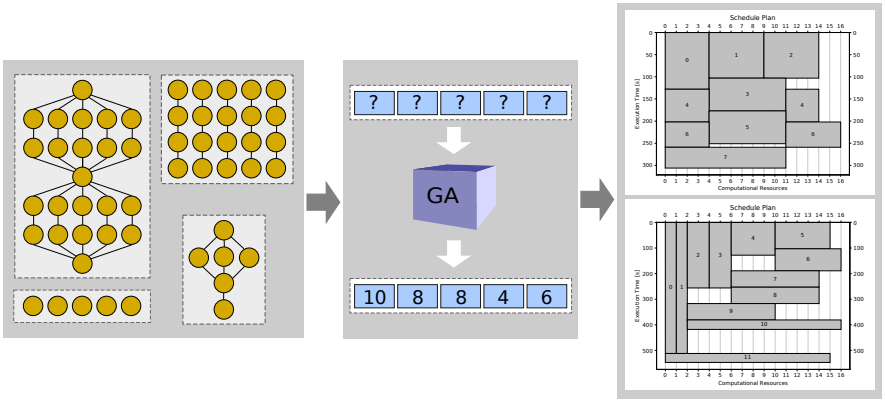

Fig. 1. A workflow is transformed to a vector of integer elements specifying assigned amount of resources to particular tasks. This vector represents a candidate solution of the GA search space. The final output of the optimization can be visualized as a workflow schedule.

the execution time for every task is calculated based on the task type, execution parameters set by the GA, input data size, and known parallel efficiency/strong scaling behavior. Next, the tasks are submitted to the cluster simulator that draws up an execution schedule and calculates the makespan (the critical path through the workflow including queuing times) and execution cost. The output of the optimization is a set of best execution parameters for individual tasks minimizing given criteria implemented by the fitness function.

2.1 Solution Encoding

In order to optimize workflow execution schedules using GA, it is necessary to transform the workflow into a template for candidate solutions (chromosomes) I. The workflow's DAG is traversed in a breath-first manner producing a vector of N tasks (genes). Every gene i corresponds to a single task and holds the execution parameters (resources) R_i assigned to the task i, see Eq. (1).

$$I = (R_1, R_2, \ldots, R_N) \tag{1}$$

The execution parameters being investigated in this study only consider the number of computing nodes assigned to a given task. This set can be simply extended in the future to support, e.g., node cpu frequency and power cup or the number of processes/threads per node.

The number of nodes assignable to a given task is naturally constrained by 1 from the bottom, and by the size of the computing system from the top. Moreover, it is also limited by the type of the task, its scalability, and the size of input data. The strong scaling, parallel efficiency and scalability were measured for each task type and input data size in advance using short benchmark runs

and stored in the performance database. These constraints are imposed at the beginning of the fitness function evaluation.

2.2 Fitness Function

This paper considers three different types of the fitness function looking at the optimization problem from different angles: (1) Independent local optimization of the execution time for particular tasks useful when running small tasks on large HPC systems, (2) Global optimization of the whole workflow minimizing the execution time and related computational cost under on-demand resource allocations, (3) Global optimization of the whole workflow time and cost on statically allocated cluster parts, i.e., the idling nodes also contribute to the computational cost.

Local optimization of workflow tasks. This fitness functions optimizes each task independently considering only the execution time while neglecting the computational cost, see Eq. (2). This fitness function does not use the cluster simulator but only sums the execution time of all tasks. Let us note that the highest possible number of computing nodes may not lead to the fastest execution time due to unbalanced local decomposition, high overhead of parallel computation, etc.

This fitness function relies on the cluster scheduler to assemble a good execution schedule of the whole workflow when provided optimal setup for particular tasks. This statement is likely to be valid for large HPC clusters with hundreds of nodes and tasks employing low tens of nodes. From the scheduling point of view, this fitness function is the fastest one.

$$fitness = t = \sum_{i=1}^{N} t_i(R_i) \tag{2}$$

where t is the aggregated net execution time of N tasks in the workflow, each of which running on R_i nodes for time t_i.

Global optimization with on-demand allocation. This fitness function minimizes the overall execution time t of the workflow given by the sum of the execution time of the tasks along the critical path in the workflow graph (makespan [12]), together with the computational cost c given by a sum of computational cost of all tasks in the workflow, see Eq. (5).

As we know from the problem definition, those two requirements usually go against each other. Therefore, an α parameter to prioritize either makespan or cost is introduced. In order to balance between proportionally very different criteria, a kind of normalization is introduced. The makespan is normalized by the maximum total execution time of the workflow t_{max}, which is considered to be the sum of the execution times of all N tasks executed by only a single computation node in a sequential manner. The cost is normalized by the minimum

execution cost which is the cost of the workflow computed by a single node in a sequential manner, see Eq. (3). This presumption is valid for typical parallel algorithms with sub-linear scaling, i.e., parallel efficiency as a function of the number of nodes is always smaller than 1, $E(P) < 1$.

$$c_{min} = t_{max} = \sum_{i=1}^{N} t_i(1) \tag{3}$$

$$c_i = t_i(R_i) \cdot R_i \tag{4}$$

$$fitness = \alpha \cdot \sum_{j \in M} (\frac{t_j(R_j)}{t_{max}}) + (1 - \alpha) \cdot \sum_{i=1}^{N} (\frac{c_i(R_i)}{c_{min}}), \quad \text{where } M = \{i | i \in \text{CriticalPath}\} \tag{5}$$

This fitness function suits best the workflow being executed in environments with shared resources where only truly consumed resources are paid for, e.g., shared HPC systems.

Global optimization with static allocation. The last fitness function described by Eq. (8) also minimizes the workflow makespan, but the computational cost now takes into the consideration also idling nodes. Let us imagine we have a dedicated portion of the cluster consisting of 64 nodes statically allocated before the workflow has started. The computational cost, the user will be accounted for, equals to the size of the allocation multiplied by the makespan, no matter some nodes are not being used for the whole duration of the workflow execution. The fitness function thus attempts to shake down the tasks to minimize the amount of idling resources while still minimizing the makespan. The execution cost is then normalized by the highest possible cost in the dedicated system where only one node works.

$$c_{max} = t_{max} \cdot P \tag{6}$$

$$c = \sum_{i=1}^{N} t_i(R_i) \cdot R_i \tag{7}$$

$$fitness = \alpha \sum_{j \in M} (\frac{t_j(R_j)}{t_{max}}) + (1 - \alpha) \frac{c_{max} - c}{c_{max}}, \quad \text{where } M = \{i | i \in \text{CriticalPath}\} \tag{8}$$

Similarly to the previous case, t is the overall execution time of the workflow, and t_{max} is the maximum overall execution time obtained for a serial scheduling of sequential tasks. The number of nodes statically allocated to the workflow is denoted by P. The number of nodes assigned per tasks i is R_i. The maximum possible cost is represented by c_{max} while the actual cost based on the current execution parameters and the workflow structure is denoted by c.

2.3 Cluster Simulator

In order to create a workflow execution schedule and calculate the makespan, we developed a simple cluster simulator called *Tetrisator*. The name of this component is inspired by the Tetris game [5] since there is a strong analogy in arranging the blocks of different sizes and shapes with the optimization of the execution schedule to minimize execution time and cost. The blocks can be seen as tasks and their sizes are given by required amount of resources and corresponding execution time. The blocks a.k.a tasks may be molded to be "wider" or "longer" by changing the number of resources, however, their surface does not have to stay constant due to varying parallel efficiency.

Tetrisator simulates the operation of an artificial HPC system with a predefined number of computing nodes P. The tasks are submitted to the simulator in the same order as defined in the chromosome (a breadth-first top down traversal). During the submission, the numbers of nodes assigned to particular tasks are taken from the chromosome and the corresponding execution times are located in the performance database. The breadth-first traversal also allows a simple definition of task dependencies the simulator has to obey. If there are multiple tasks being ready to be executed, the submission order is followed. This is inspired the default behaviour of the PBS job scheduler with the backfilling policy switched off [27].

3 Experimental Results

The experiments presented in this paper have the following goals: (1) confirm the hypothesis that it is possible to find suitable schedules for given workflows using genetic algorithms, (2) investigate the suitability of the α parameter to prefer of one optimization criterion over the other one (overall execution time vs. computational cost), and (3) evaluate the computational requirements of the optimization process on various workflow sizes.

3.1 Investigated Moldable Workflows

The performance and search capabilities of the proposed optimization algorithm were investigated on three scientific workflows inspired by real-world applications of the acoustic toolbox k-Wave [19] for validation of neurostimulation procedures, see Fig. 2. The workflows are composed of two different kinds of tasks, simulation tasks (ST) and data processing tasks (PT). The first workflow shows a barrier behaviour where all simulation tasks at the first level have to finish before the data is processed by a single data processing task. Only after that, the second level of STs can continue. The second workflow uses a reduction tree where the data processing is parallelized in order to reduce the execution time of PT tasks. The last workflow, not shown in the figure, is composed of the set of independent STs executed in embarrassingly parallel manner.

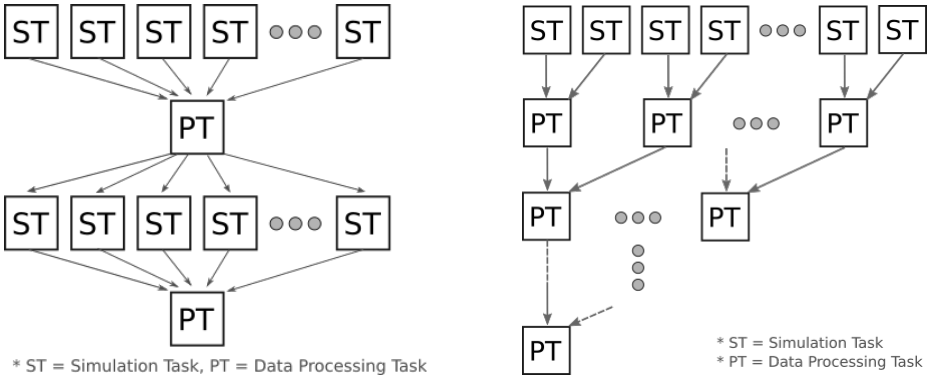

Fig. 2. The structure of two investigated workflows. The simulation tasks are interleaved with data processing tasks implying barriers between stages (left), the data produced by the simulation tasks are merged via a reduction tree (right).

Fig. 3. Strong scaling of the k-Wave toolbox measured for $< 1, 36 >$ nodes on domain sizes composed of 500^3, 512^3 and 544^3 grid points. k-Wave simulations are the main part of simulations tasks in the examined workflows.

The simulation tasks are heavy computing programs scalable from 1 to 36 nodes, see Fig. 3. Their scaling was measured using the C++/MPI implementation of the k-Wave toolbox on the Barbora supercomputer at IT4Innovations[1]. The scaling behaviour depends on the input data size and shows several local optima for the number of nodes being powers of two. The shortest execution time was seen for 35 nodes. From these three examples, the first one was chosen in our experiments. The data processing tasks are lightweight tasks executable on one or two nodes. Their time complexity grows linearly with the number of input files they have to process.

In real k-Wave applications, the size of the domain for all STs is the same, however, the amount of time steps may vary by up to 25%. This is given by

[1] https://docs.it4i.cz/barbora/introduction/

the mutual position of the transducer and the patient's head, which influences the distance the ultrasound wave has to travel. Moreover, the performance of all processors in the cluster is not equal. According to [1], the fluctuations may cause up to 5% deviations in the execution time. Both factors are considered by adding random perturbations to the task execution time during the workflow generation.

3.2 Local Task Optimization of the Execution Time

First, we investigated the local optimization of the proposed workflows, which is the simplest kind of optimization. This optimization only considers the net execution time of all tasks neglecting the queueing times and simulation cost. Thus no α parameter is used. This technique shows very good capabilities in optimizing particular tasks. From 20 independent runs of the GA, more than 90% of trials always found the best possible solution, the fitness of which can be analytically derived.

Table 1 shows suitable parameters for the genetic algorithm along with the number of generations necessary to find the optimal schedule, the execution time in seconds and the success rate. Since the variability of the results across different workflows was negligible, we collapsed all results into a single table.

The table reveals that the necessary population size linearly grows with the size of the workflow from 25 up to 150 individuals, but still stay quite small. This is natural behaviour since bigger workflows require longer chromosomes which in turn requires larger populations to keep promising building blocks of the solution. The best selection strategy driving the GA through the search spaces appears to be Steady state selection, although the difference to the Rank and Tournament selections was marginal. The number of generations to be evaluated before the GA finds the optimal schedule stays relatively constant close to 200. On the other hand, the execution time appears to grow quadratically. This growth can be attributed to a product of increasing population size which rises the number of fitness function evaluations, and the linearly growing time complexity of the fitness function evaluation. Nevertheless, an execution time of 14s with 95% of success rate for the biggest workflow is an excellent result.

3.3 Global Workflow Optimization of Execution Time and Cost

The global optimization of the workflow considers both criteria and balances between them using the α parameter. In this section we investigate two fitness functions oriented on on-demand and statically allocated resources.

The influence of the α parameter. Let us first investigate the influence of the α parameter on both global fitness functions. In practise, the α parameter can be seen as a user-friendly control slider promoting either the execution time or cost. The following values of α were tested: 0.95 and 0.8 prioritizing the minimal makespan, 0.5 balancing the makespan and the cost / usage of resources),

Table 1. Computation requirements of the local optimization method together with recommended genetic algorithm (GA) settings that lead to optimal schedules obtained in the shortest time. Other GA settings common for all experiments is uniform crossover of 0.7 probability, random mutation of 0.01 probability, and 5% elitism.

Workflow Size	Population Size	Selection Method	Median Number of Generations	Average Runtime	Success Rate
7	25	Steady State, Rank	180	0.27s	100%
8	25	Steady State, Rank	220 - 250	0.37-0.42s	100%
15	50	Steady State, Rank, Tournament, Roulette Wheel	120-200	0.52-0.87s	100%
16	50	Steady State, Rank, Tournament	180-200	0.98 - 1.08s	100%
31	100	Steady State, Rank	100	1.40s	100%
32	100	Steady State, Rank	190	3.41s	90-95%
63	100	Rank, Steady State	215	5.61s	100%
64	150	Steady State, Rank	260	13.29s	90-95%

and 0.2 and 0.05 prioritizing the minimal execution cost and unused resources, respectively.

For each value of α and suitable GA settings, 20 independent runs were carried out. For the sake of brevity, only a few examples of selected workflows with the best GA settings will be shown. For each example, the results from all runs were collected, sorted, and 5% of the best solutions visualized in the form of a Pareto frontier. The color of the data points and lines representing the frontiers correspond to the α parameter used.

Let us start with the quality of solutions produced by the on-demand allocation fitness, see Fig. 4 and 5. Although evolved solutions for various α parameter may slightly overlap, Fig. 4 shows that we managed to drive the genetic algorithm to find desired solutions (forming clusters) that meet given optimization constraints. By adjusting the α parameter, we move along an imaginary curve composed by the combination of all Pareto frontiers. Thus when the importance is attached to the simulation cost, it is possible to get a schedule that reduces the cost by 10%, however, runs for 12% longer time, and vice versa. It can also be seen that each value of α works well only in a relatively short interval (the middle of the frontier). At the edges it is usually outperformed by a different values of α.

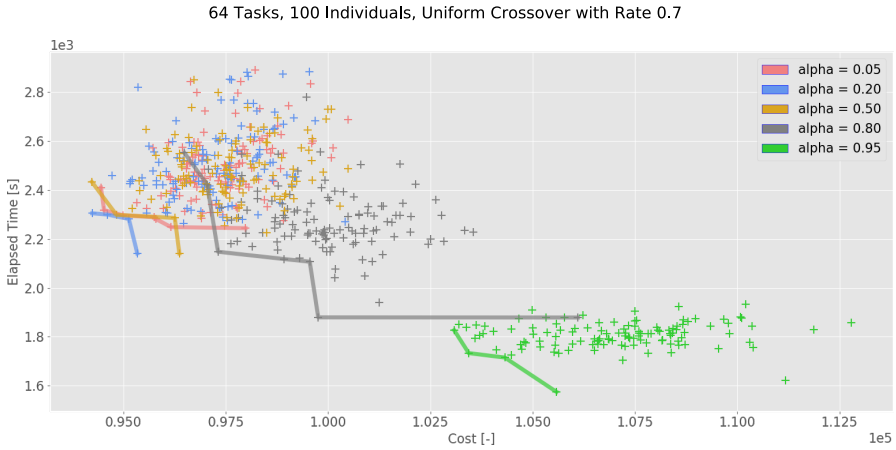

Fig. 4. Pareto frontier and dominated solutions calculated using the fitness function for on-demand allocations for the workflow with two levels of simulation tasks and various values of the α parameter.

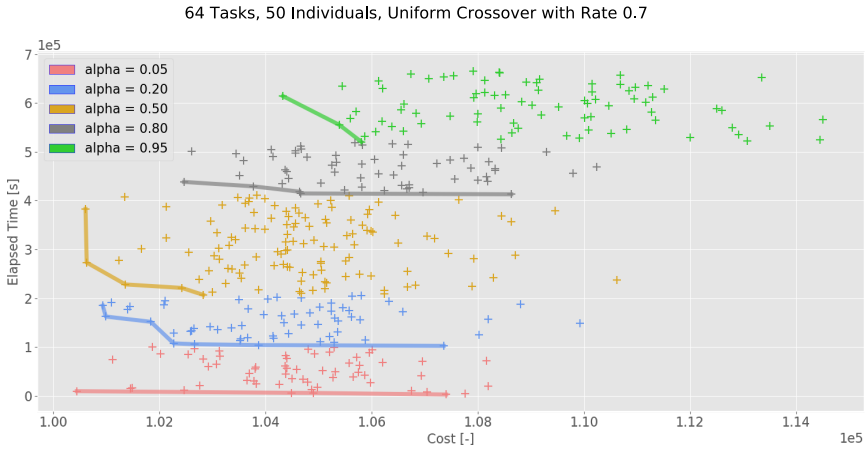

Fig. 5. Pareto frontier and dominated solutions calculated using the fitness function for on-demand allocations for the workflows without dependencies and various values of the α parameter.

The workflows without dependencies, however, show much worse parametrization, see Fig. 5. The only sensible values of α seem to be 0.05. Other values produce much worse compromises between time and cost. The only exception is the value 0.95 which can offer 15% - 20% cost-effective schedules, but many times slower.

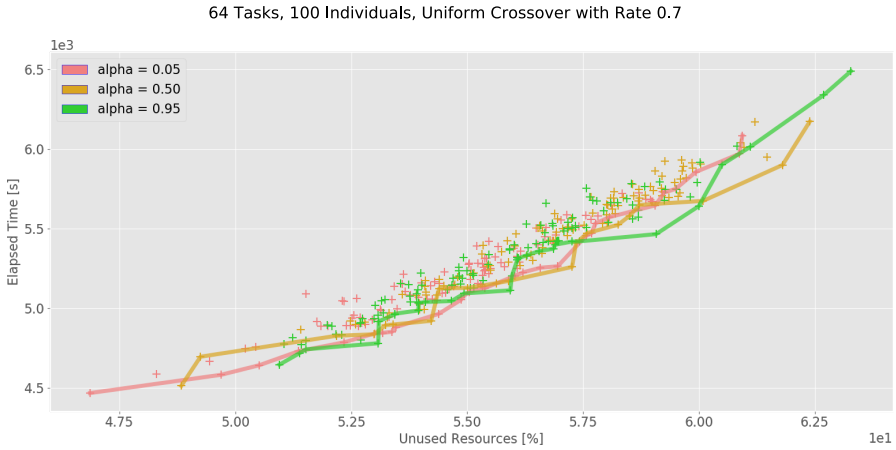

Fig. 6. Pareto frontier and dominated solutions calculated using the fitness function for static allocations for the workflow with two levels of simulation tasks and various values of α balancing between makespan and percentage of unused resources.

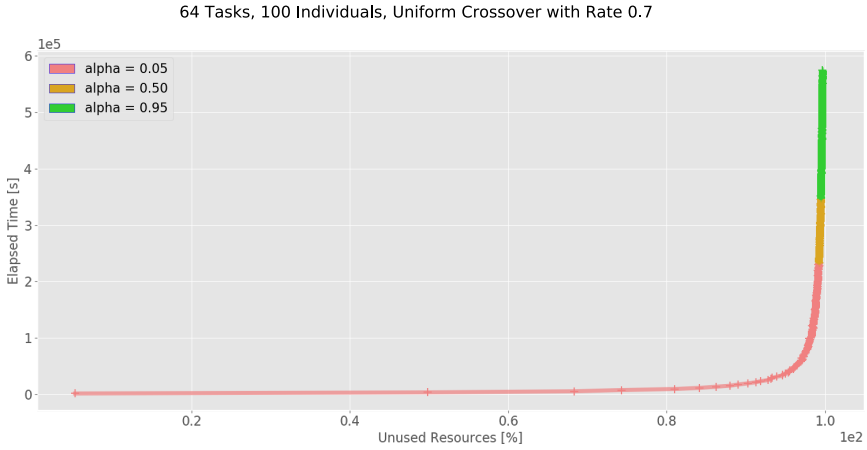

Fig. 7. Pareto frontier and dominated solutions calculated using the fitness function for static allocations for the workflow without dependencies and various values of α balancing between makespan and percentage of unused resources.

For experiments using the fitness function for static cluster allocations, we only show three different α parameters because the solutions highly overlap, see Figs 6 and 7. The solutions found for workflows containing task dependencies seem to be saturated by the same minimal execution time. Smaller α parameter pushes the genetic algorithm to find solutions with smaller amount of unused resources (up to 43%) but a range of found solutions is quite high. From this

Table 2. Recommended settings for the GA and the on-demand allocation fitness function. Other settings common for all experiments are uniform crossover of 0.7 probability, random mutation and 5% elitism.

Alpha	Workflow Size	Population Size	Selection Method	Mutation Probability	Median Number of Evaluations
0.95	7 - 16	25	Rank	0.001	2000 - 19750
	31 - 64	50	Steady State	0.01	5000 - 10000
0.80	7 - 16	25	Steady State	0.01	1250 - 2500
	31 - 64	50, 100	Rank	0.001	22000 (50) - 45000 (50)
0.50	7 - 64	50, 100	Rank	0.001	2500 (50) - 65000 (100)
0.20	7 - 64	25, 50	Rank	0.001	8750 (25) - 42500 (50)
0.05	7 - 64	50, 100	Steady State	0.01	5000 (50) - 30000 (100)

point of view, 0.05 for α gives the most reasonable solutions. This is even more visible for workflows without task dependencies where 0.05 for the α parameter optimizes both makespan and the amount of unused resources.

The experiments showed that both criteria, the makespan and percentage of idle resources, are highly correlated. Thus, the lower percentage of idle resources the faster execution time. Although this may sound natural, the anomalies in the scaling behaviour of particular tasks has the potential to break this presumption. This experiment, however, shows that the scaling plots are very close to the perfect scaling.

Suitable Parameters of the Genetic Algorithm. Table 2 presents recommended settings for the genetic algorithm which produced best results along with the computational requirements expressed as the number of fitness function evaluations (i.e., a product of the number of generations and the population size). When two population sizes are given, the smaller one is used for the smaller workflows, and vice versa. The median number of evaluation is calculated for the actual population size shown in bracket in the last column. The range is bounded by two values, the one for the smallest workflow in the range and the one for the biggest workflow.

Table 3 presents an average execution time for a single generation. In connection with Table 2, the absolute wall clock time of the evolution process can be calculated. As an example, a schedule for a workflow with 64 dependant tasks can be evolved in 2 minutes and 20 seconds. We found out that schedules for tasks without dependencies may be evolved in 2 to 3 times shorter time.

The recommended settings of the genetic algorithm covers 0.7 probability of uniform crossover, steady state selection, 1% random mutation and 5% elitism. Workflow with less than 31 tasks could be evolved with 25 individuals in the population whereas bigger workflows (up to 64 tasks) with 50 individuals. It took approximately from 2500 (100 generations for 25 individuals) to 25000 (500 generations for 50 individuals) evaluations to evolve schedules for workflows of

Table 3. The execution time of the evolution process for various workflows with dependencies and population sizes measured using global fitness functions on the Salomon cluster at IT4Innovations. The evolution runtimes for workflow without dependencies are approximately three times smaller.

Population Size	Runtime per a Single Generation in Seconds							
Workflow Size	7	8	15	16	31	32	63	64
25	0.004	0.005	0.010	0.010				
50	0.007	0.009	0.019	0.021	0.040	0.043	0.112	0.120
100	0.013	0.018	0.037	0.043	0.077	0.088	0.227	0.220
150					0.110	0.132	0.382	0.335

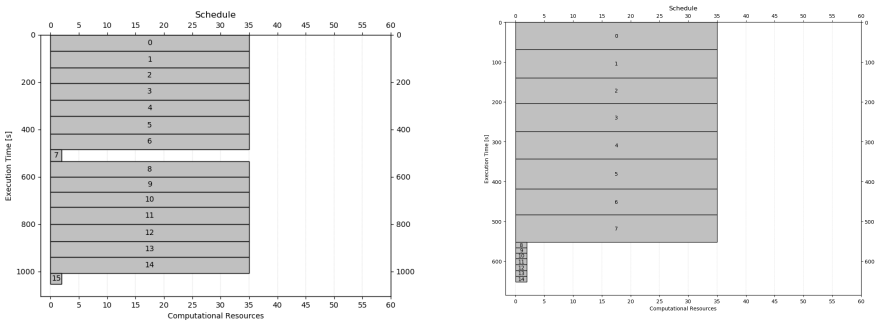

Fig. 8. Evolved schedules for investigated workflows with 16 and 15 dependant tasks using a local optimization.

7 to 64 tasks. So, a schedule for the workflow of 64 tasks with dependencies is evolved in a minute.

Investigation of the Workflow Schedules. Here, we show and compare several evolved schedules using different fitness functions. For better visibility, only schedules for workflows of 15 and 16 tasks are shown. Figure 8 shows two execution plans for 15 and 16 tasks, respectively, locally optimized by Eq. (2). Regardless of the number of tasks in the workflow, the genetic algorithm always picks 35 nodes for simulation tasks and 2 nodes for data processing tasks because this selection assures their minimal execution time.

Figure 9 shows evolved schedules using a global optimization for on-demand allocations balancing the makespan and computational cost with $\alpha = 0.5$. When compared with schedules depicted in Fig. 8, it is evident that the GA preferred much smaller amounts of nodes for simulation tasks which resulted in a cost reduction by 40% and makespan increase by 37% for the workflow of 16 tasks. In the case of the workflow with 15 tasks, we may however observe that the makespan is even better when the global optimization is used. This is given by the way the local optimization works, i.e., the concurrency is not expected. Here,

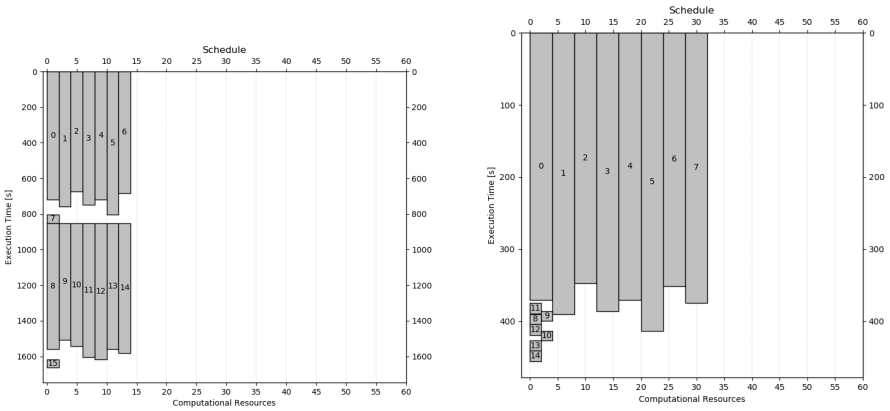

Fig. 9. Evolved schedules for investigated workflows with 16 and 15 dependant tasks using a global optimization for on-demand allocations balancing the makespan and computational cost.

the global optimization gives better results in both aspects, i.e., the makespan was reduced by 30% and the computational cost by 37%.

Figure 10 shows evolved schedules using a global optimization balancing the makespan and the amount of unused resources with $\alpha = 0.5$. If we compare them with schedules in Fig. 8, we can see that in both cases the makespan is reduced by 34% in case of 15 tasks, and by 19% in case of 16 tasks while the amount of unused resources was reduced from 53.0% and 50.0% to 30.8% and 38.2%, respectively.

Since the global optimization approaches are not comparable, we just emphasize the differences between obtained solutions in Fig. 9 and Fig. 10. It can be seen that the solutions evolved using the global optimization focusing on the amount of unused resources have shorter makespans because there is an effort to use, i.e., pay for, the resources for the shortest time. If we evaluate these solutions using the fitness function for on-demand allocations, it is obvious that we get more expensive solutions than those which were originally evolved using this fitness function. Since we cannot compare the solutions in absolute numbers, we can come out of the premise that computational cost equals to actually used resources. In other words, the solution for 15 tasks found by fitness function for static allocations used 69% of available resources but the solution found by fitness for on-demand allocations used only 42%. The same states for 16 tasks where the solution found by the fitness function for static allocations used 62% of available resources but the solution found by the fitness function for on-demand allocations used only 19%.

The makespans of 15 tasks schedules are roughly the same (5% difference) but there is a 36% difference in obtained computational cost and the amount of unused resources.

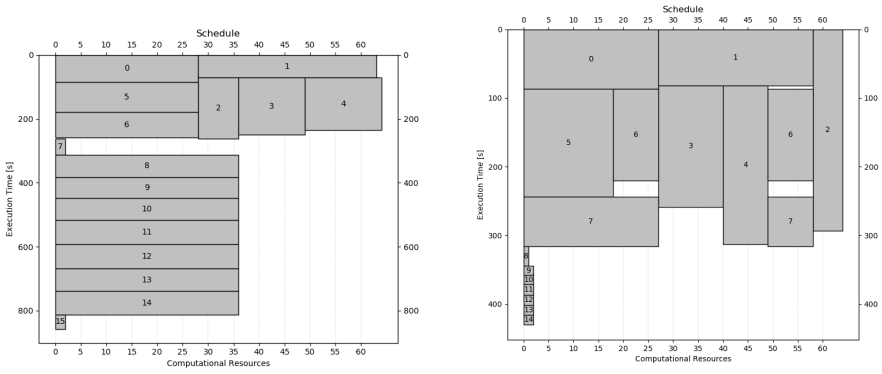

Fig. 10. Evolved schedules for workflows with 16 and 15 dependant tasks using a global optimization balancing the makespan and the amount of unused resources (37.7% for the left workflow, 30.8% for the right workflow).

4 Conclusions

This paper investigates the execution optimization of moldable scientific workflows. It uses genetic algorithm to evolve schedules for workflows comprising of two kinds of tasks with and without mutual dependencies. The presented objective functions use collected historical performance data for supported workflow's tasks. Those objective functions implement a trade-off coefficients that allow the schedule customization to either minimize one objective to another or to balance them. The paper introduces three objective functions that provide the (1) local optimization of workflow tasks minimizing their execution times, (2) global optimization with on-demand resource allocation balancing the workflow makespan and its computational cost, and (3) global optimization with static resource allocation balancing the workflow makespan and the cluster's idling nodes.

After performing the experiments, we confirmed our hypothesis that (1) we are able to generate good schedules for various workflows as well as meet different optimization criteria. For local optimization, we got very good results where more than 90% of performed trials found the optimal solution. (2) When performing a multi objective optimization, we introduced an α trade-off parameter and confirmed we can prioritize one objective to another. Here, we got the best results for the global optimization with on-demand resources and workflows with task dependencies where solutions found for the different parameter α form clusters. Let us note, that the trade-off parameter allows to customize the solution parameters only in a limited scale of makespan and cost, e.g., 10%. Much worse parametrization could be seen for workflows without task dependencies for both global optimization methods where the only value 0.05 of α produced sensible solutions. (3) We measured and summarized computational demands of each

presented objective function and workflows of different sizes. Using the local optimization, the workflow of 64 tasks could be evolved in 14 seconds. Global optimization is more computationally demanding but we managed to get the schedule for the most complex workflow with task dependencies in roughly 2 minutes. Finally, the paper provides the genetic algorithm settings to reproduce the presented results.

4.1 Future Work

Here we summarize several ideas to be addressed soon. First, we would like to better tune the presented trade-off coefficient α and better define ourselves among other already existing optimization heuristics. Next, we would like to validate our approach against standard task graphs[2] of different sizes.

Furthermore, a couple of the algorithm improvements is to be addressed. Currently used cluster simulator traverses tasks within a workflow in a breadth-first top down order and follows the task submission order when multiple tasks are ready to be executed. A mechanism such as backfilling commonly presented in the PBS job scheduler is not implemented. In practise, we use mainly PBS-based clusters for our workflows, thus, we would like to integrate this functionality to the presented Tetrisator. We will also consider a possibility to integrate an already existing cluster simulator, e.g., ALEA [21].

Next, more real world tasks together with their measured performance data would be incorporated. In reality, we usually cannot measure and hold performance data for all input data sizes and input parameters options. Therefore, we need to implement interpolation based heuristics [20].

Acknowledgments. This project has received funding from the European Union's Horizon 2020 research and innovation programme H2020 ICT 2016-2017 under grant agreement No 732411 and is an initiative of the Photonics Public Private Partnership.

References

1. The shift from processor power consumption to performance variations: fundamental implications at scale. *Computer Science - Research and Development*, 31(4):197–205, 2016.
2. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 1820 1967 spring joint computer conference*, 23(4):483–485, 1967.
3. S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*, pages 1–10. IEEE, nov 2008.
4. R. Bleuse, S. Hunold, S. Kedad-Sidhoum, F. Monna, G. Mounie, and D. Trystram. Scheduling Independent Moldable Tasks on Multi-Cores with GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2689–2702, sep 2017.

[2] http://www.kasahara.cs.waseda.ac.jp/schedule/

5. R. Breukelaar, E. D. Demaine, S. Hohenberger, H. J. Hoogeboom, W. A. Kosters, and D. Liben-Nowell. Tetris is hard, even to approximate. In *International Journal of Computational Geometry and Applications*, 2004.
6. T. F. Chan and T. P. Mathew. Domain decomposition algorithms. *Acta Numerica*, 1994.
7. A. M. Chirkin, A. S. Belloum, S. V. Kovalchuk, M. X. Makkes, M. A. Melnik, A. A. Visheratin, and D. A. Nasonov. Execution time estimation for workflow scheduling. *Future Generation Computer Systems*, 75, 2017.
8. C. Cotta and A. J. Fernández. *Evolutionary Scheduling*, volume 49 of *Studies in Computational Intelligence*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
9. P.-F. Dutot, M. A. S. Netto, A. Goldman, and F. Kon. Scheduling Moldable BSP Tasks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 157–172. 2005.
10. D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 1–26. 1996.
11. A. F. Gad. Geneticalgorithmpython: Building genetic algorithm in python. `https://github.com/ahmedfgad/GeneticAlgorithmPython/tree/05a069abf43146e7f8eb37f37c539523bf62ac9a`, 2021.
12. R. Hejazi, S., and S. Saghafian. Flowshop-scheduling problems with makespan criterion: a review. *International Journal of Production Research*, 43(14):2895–2929, jul 2005.
13. R. L. Henderson. Job scheduling under the Portable Batch System. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 279–294. 1995.
14. M. D. Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4):18–21, dec 1990.
15. M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in HPC resource management systems: Queuing vs. planning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2862(June):1–20, 2003.
16. K.-C. Huang, T.-C. Huang, M.-J. Tsai, and H.-Y. Chang. Moldable Job Scheduling for HPC as a Service. In *Lecture Notes in Electrical Engineering*, pages 43–48. 2014.
17. H. Izadkhah. Learning based genetic algorithm for task graph scheduling. *Applied Computational Intelligence and Soft Computing*, 2019, 2019.
18. K. Jansen and F. Land. Scheduling Monotone Moldable Jobs in Linear Time. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 172–181. IEEE, may 2018.
19. J. Jaros, A. P. Rendell, and B. E. Treeby. Full-wave nonlinear ultrasound simulation on distributed clusters with applications in high-intensity focused ultrasound. *International Journal of High Performance Computing Applications*, 30(2):1094342015581024–, 2015.
20. M. Jaros, T. Sasak, E. B. Treeby, and J. Jaros. Estimation of execution parameters for k-wave simulations. In *High Performance Computing in Science and Engineering*, pages 116–134. Springer International Publishing, 2020.
21. D. Klusacek, G. Podolnikova, and S. Toth. Complex job scheduling simulations with alea 4. In G. Tan, editor, *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, pages 124–129, Belgium, 2016. ICST.
22. A.-L. Lamprecht and K. J. Turner. Scientific workflows. *International Journal on Software Tools for Technology Transfer*, 18(6):575–580, nov 2016.

23. A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.
24. F. A. Omara and M. M. Arafa. Genetic algorithms for task scheduling problem. *Journal of Parallel and Distributed Computing*, 70(1):13–22, 2010.
25. Y. Robert, S. Shende, A. D. Malony, A. Morris, W. Spear, S. Biersdorff, B. Smith, D. Wang, D. Ricciuto, W. Post, M. W. Berry, F. Irigoin, K. Yelick, S. L. Graham, P. Hilfinger, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, T. Wen, J. Dongarra, P. Luszczek, A. Bhatele, S. M. Freudenberger, V. Diekert, A. Muscholl, M. Herlihy, and J. E. B. Moss. Task Graph Scheduling. *Encyclopedia of Parallel Computing*, pages 2013–2025, 2011.
26. Srinivasan, Krishnamoorthy, and Sadayappan. A robust scheduling technology for moldable scheduling of parallel jobs. In *Proceedings IEEE International Conference on Cluster Computing CLUSTR-03*, pages 92–99. IEEE, 2003.
27. S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective Reservation Strategies for Backfill Job Scheduling. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2537 of *LNCS*, pages 55–71. Springer Verlag, 2002.
28. D. Sudholt. Parallel Evolutionary Algorithms. In *Springer Handbook of Computational Intelligence*, pages 929–959. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.
29. D. Ye, D. Z. Chen, and G. Zhang. Online scheduling of moldable parallel tasks. *Journal of Scheduling*, 21(6):647–654, dec 2018.
30. A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 44–60. 2003.