Multi-Objective Optimization of Real-Time Task Scheduling Problem for Distributed Environments

Salimi, Maghsood; Majd, Amin; Loni, Mohammad; Seceleanu, Tiberiu; Seceleanu, Cristina; Sirjani, Marjan; Daneshtalab, Masoud; Troubitsyna, Elena

Published in:
Proceedings of the 6th Conference on the Engineering of Computer Based Systems

DOI:
10.1145/3352700.3352713

Published: 01/01/2019

Document Version
Accepted author manuscript

Document License
Publisher rights policy

Multi-objective Optimization of Real-Time Task Scheduling Problem for Distributed Environments

***Abstract*—Real-world applications are composed of multiple tasks which usually have intricate data dependencies. To exploit distributed processing platforms, task allocation and scheduling, that is assigning tasks to processing units and ordering inter-processing unit data transfers, plays a vital role. However, optimally scheduling tasks on processing units and finding an optimized network topology is an NP-complete problem. The problem becomes more complicated when the tasks have real-time deadlines for termination. Exploring the whole search space in order to find the optimal solution is not feasible in a reasonable amount of time, therefore meta-heuristics are often used to find a near-optimal solution.**

We propose here a multi-population evolutionary approach for near-optimal scheduling optimization, that guarantees end-to-end deadlines of tasks in distributed processing environments. We analyze two different exploration scenarios including single and multi-objective exploration. The main goal of the single objective exploration algorithm is to achieve the minimal number of processing units for all the tasks, whereas a multi-objective optimization tries to optimize two conflicting objectives simultaneously considering the total number of processing units and end-to-end finishing time for all the jobs. The potential of the proposed approach is demonstrated by experiments based on a use case for mapping a number of jobs covering industrial automation systems, where each of the jobs consists of a number of tasks in a distributed environment.

***Index Terms*—Distributed Task Scheduling, Real-Time Processing, Evolutionary Computing, Multi-Objective Optimization**

I. Introduction

Industrial applications often require guaranteeing real-time execution, fault tolerant implementations and providing reliable functionality. In general, it is impossible for a single processing unit to satisfy all these needs. However, a distributed processing environment provides a variety of computational capabilities, which can be utilized to perform an application that has diverse execution requirements. An application job can be decomposed into tasks. Tasks may have data dependencies and it is possible that each task needs a certain computational throughput. For distributing tasks, the following decisions should be made respectively: ① task allocation, i.e. assigning tasks to processing units, and ② tasks scheduling, i.e. defining task execution order and the order of data transfers among processing units. The general goal of task allocation and scheduling is to minimize the end-to-end cost of computation, i.e. minimizing overall response time of the application, minimizing the number of processing units, or both.

Performance of such parallel systems can be optimized by employing an efficient task allocation and scheduling approach, however, the allocation and scheduling problem is an NP-complete problem [1]. Using exhaustive approaches for finding optimal solution is time-consuming and is impossible in practice. Many heuristic task scheduling strategies have been proposed [2], [3] to find a near-optimal solution in a reasonable amount of time. Evolutionary Computing (EC) is a set of methods proposed to solve the allocation and scheduling problem. Genetic Algorithm (GA) is a popular EC method which can better locate a near-optimal solution than other similar approaches in most cases [4]–[7]. Although GA is a powerful solution, defining a proper fitness function is always challenging and requiring expertise especially when the size of design space is huge. Plus, GA is relatively slow and may be trapped in local optima [8].

To overcome aforementioned challenges, a Multi-Population Genetic Algorithm (MPGA) [9] is leveraged in this research for task allocation and scheduling over a collection of nonuniform processing units. MPGA is a static scheduling strategy, where the execution times of tasks and the data transfer times between tasks are known. MPGA is the parallel version of GA that provides better convergence rate and more speedup compared to single population GA [8]. In addition, MPGA highly reduces the probability of falling into local optima trap. Two different MPGA strategies have been considered to solve the allocation and scheduling problem including: ① *Single objective optimization* and ② *Multi-objective optimization*. While the single objective optimization minimizes the number of processing units, the multi-objective optimization considers the second conflicting metric, jobs end-to-end finishing time, to find solutions satisfying multiple user needs.

Contribution. In a nutshell our main contributions are:

- In this paper, we solved task allocation and scheduling problem in a distributed environment. To attain this purpose, we leveraged a MPGA optimization method with different optimization scenarios.
- Defining novel fitness functions to efficiently explore the design space in both single and multi-objective optimization scenarios.
- The evaluation results based on an industrially inspired use case show the impact of the proposed fitness function while converging to better solutions.

Paper Organization. The paper is organized as follows. Section II defines the allocation and scheduling problem and our use case. Section III explains the MPGA and the specifications of fitness functions for both single objective and multi-objective optimization scenarios. Section IV presents the experimental results and demonstrates the efficiency and

convergence of the proposed algorithm. Some related work reviewed in Section V. We end with concluding remarks and future work in Section VI.

II. Problem Definition

We start here by describing a generic distributed process control system. In such systems, a series of computing devices operate on data collected from sensors placed close to a physical process, and update control signals to other devices - actuators - able to control the evolution of the process. A process may be exemplified by a simple tank-filling operation or by more complex systems, such as ore separation, water purification, etc. The process parameters (such as liquid levels, temperatures, etc.) are usually required to be maintained within a certain range of values, even when the environment is disturbed. Whenever new values are presented via sensors to the processing devices, certain procedures hosted within these devices are launched, and potential new values are sent to the process-responsible actuators. In large systems, there are potentially thousands or more such procedures, installed in tens to hundreds of *control devices*.

The main problem that we raise here is how to allocate the number of processing operations on an as small as possible set of processing devices, such that planned operations are not affected with respect to their timing and duration, and the processing devices are operating within their nominal characteristics.

In order to cover most of the aspects of interest when solving this, in the following we employ a synthetic example of a system as use case, with elements presented in Fig. 1. Here, we have a control system composed of 8 jobs, their characteristics and further decomposition being detailed below.

A. System Model Elements

The system we consider is composed of a number of complex control processes, referred from now on as *jobs*. The system reads data from a set of input elements - the *sensors* S1,...,S11 and processes the data on a number of available *processing units* (P1,..., P24). The processed data is sent further in the system to other elements - the *actuators* A1,...A11 - notice that having a similar number of sensors and actuators is a coincidence of no relevance in the analysis to come. A job refers to the data trip from sensors to actuators.

A job can be further described as a collection of *tasks*, acting mostly sequentially, but not excluding parallel processing - especially if tasks belong to different jobs. To illustrate a more critical situation, we assume that all the considered tasks are non-interruptible. A task is a unitary, and with the assumed non-interruptible characteristic, an *atomic* system element, to be executed on one of the available processing units. Each task has input either a sensor, or the output of a precedent task. At the output of a task stays either an actuator, or the input of a follower task. Multiple inputs to a task are possible (see task T13 in Job8 on actuator A10), in which case all of them must be present for the task to start its operation. Differently, if a task has more than one output (see task T21 in Job7), all of them are presented at the same moment.

Fig. 1 describes additional information pertaining to task and job execution, as well as some characteristics of interest for the processing units. Thus, a task is also defined with a potential maximal load that it presents to the processing unit, and with a maximal execution time. For instance, task T1 in Fig. 1 produces a maximal load of 10, and it executes in maximum 10tu ("time units": μ-seconds to seconds, for instance. However, an actual specification of these units is not of interest in our work here). At the same time, the available processing unit P1 can withhold a maximum load of 100, and possess 5 connection interfaces.

In their turn, the jobs have an execution time (the sum of the execution times of the composing tasks), and a frequency: how often data is read from sensors, and it has to be sent to the actuators. For instance, Job1 - a sequence of T1, T14 and T18 - has a maximal execution time of 60tu, and it is recurring every 70tu. A more complex situation is presented by jobs 3, 7 and 8, where two actuators are related to each job. The times for processing the data corresponding to each actuator may be different, but what holds them together is the execution frequency (200tu, 400tu and 500tu, per job, respectively). Fig. 2 shows the dependency graph between tasks of a job example from the use case (see Section II.A).

B. Problem Assumption

To only focus on the allocation and scheduling problem, we made the following additional assumptions. First of all, we assumed that each task is written in a machine-independent language. Moreover, it is assumed that we know all the data dependencies among tasks before execution (as described by Fig. 1 and partially by Fig. 2). The distributed processing platform is nonuniform, consisting of multiple homogeneous processing units with various processing potential.

If a data conditional is based on input data, it is assumed to be contained inside a task. A loop that uses an input data item to determine one or both of its bounds is also assumed to be contained inside a task. When two communicating tasks are mapped onto the same processing units we assume that the communication delay is zero. However, when they are mapped onto different processors a finite communication delay is assumed and modeled by 1tu.

Moreover, we do not (yet) consider here aspects related to reliability, fault tolerance, safety, etc. These aspects may (such as in the case of fault tolerance) require a duplication of allocation and synchronization of data across duplicated locations. These additional objectives are subject of further work analysis.

III. MPGA Description

GA is an iterative population-based exploration solution mimicking the process of natural selection and evolution where the characteristics of the process can be utilized in solving optimization problems. All GA-based methods have an initial population where selection, crossover, mutation operators are applied to initial population for producing improved population. The operations will be repeated until satisfying

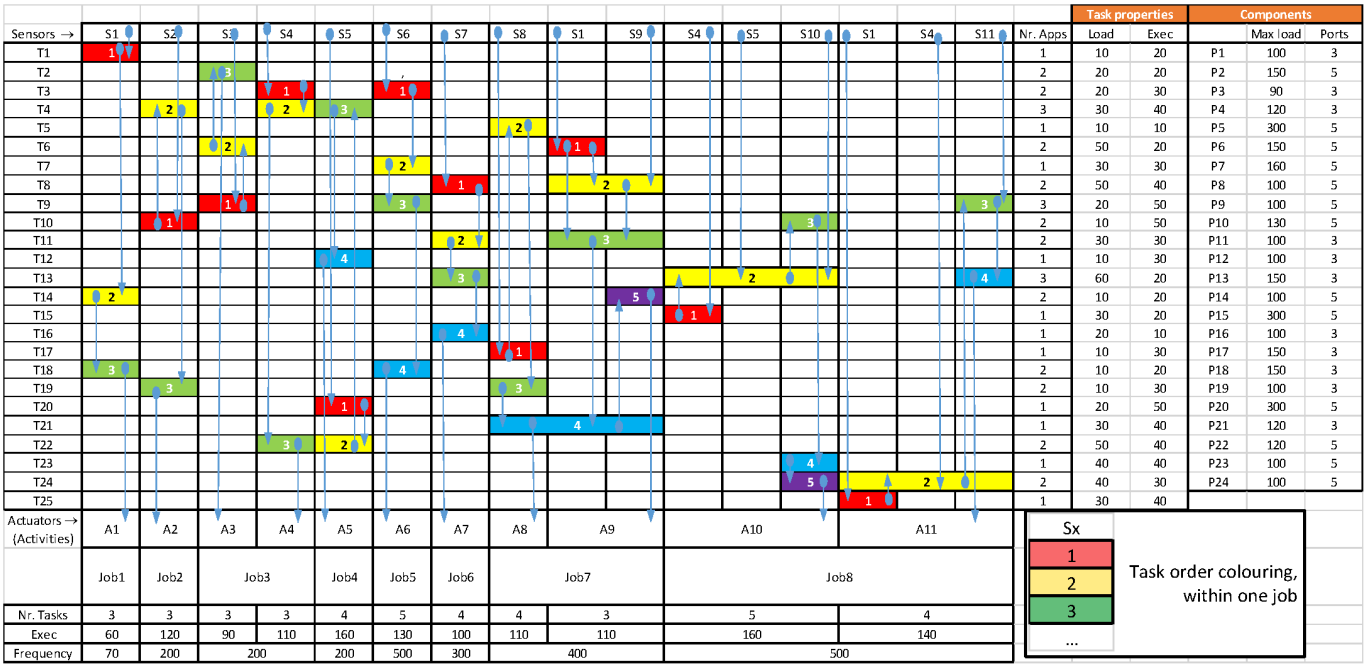

Fig. 1. Representing The use case including jobs, intra-task dependencies, tasks load complexity, real-time deadlines and processing unit specifications.

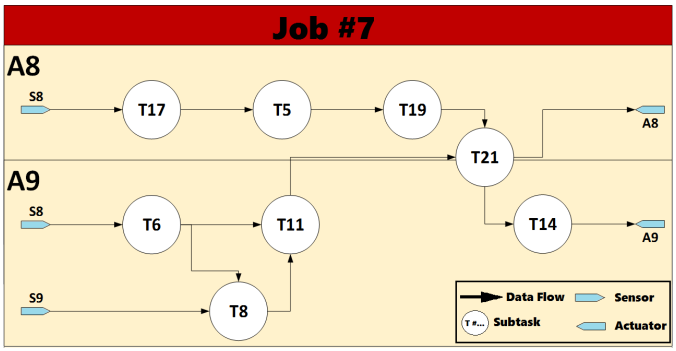

Fig. 2. Dataflow of Job #7.

user criteria (reaching suitable results) or stopping after a predefined number of iterations. The following subsections explain the basic components of GA.

Step 1. **Generating Initial Population.** The initial population includes random solutions in the design space, where each solution represented by chromosome is a scheduling for all the jobs. The size of initial population depends on the size of design space. To check the validity of solutions in the initial population, each solution is examined by using the objective function represented in Equation (1). Invalid solutions will be removed from the population.

Step 2. **Fitness Evaluation.** Objective function (fitness function) is a metric for comparing different scheduling that satisfy problem constraints. Equation (1) and Equation (2) represent the fitness functions for single-objective and multi-objective optimization, respectively.

$$Fitness\_1 = \#Processors + (\gamma \times (\alpha + \beta + \theta)) \quad (1)$$

$$Fitness\_2 = \frac{\#Processors}{\gamma} + \frac{Run - time}{BiggestDeadline} + 3 \times (\alpha + \beta + \theta) \quad (2)$$

where α is the total extra loads of the all assigned tasks that exceed the load of processing units, β is the total extra deadline of all assigned tasks that exceed the real-time deadlines, θ is the total extra ports of all job assignments that exceed the total number ports per processing units, and γ is equal to 23, the total number of processing units. $BiggestDeadline$ is the maximum possible time for finishing the slowest job. The scale of extra load (α), extra deadline (β), and extra ports (γ) could be very different, thus all the α, β, and γ parameters should be normalized. However, we did not normalized them since the range of these parameters are deterministic and the fitness functions are customized for the studied use case.

In (1), minimizing the number of processors ($\#Processor$) is the exploration objective. Whereas in (2), minimizing both the end-to-end finishing time of all the jobs ($Run-time$) and $\#Processor$ are the exploration objectives.

Step 3. **Selection.** Obviously the schedules with better fitness function are selected as the next generation and the others will be removed from population set. The goal is to find a solution in design space with lowest fitness function in both Equations (1) and (2).

Step 4. **Crossover Operator.** Is the most important operator of GA. GA randomly selects two genomes from the population

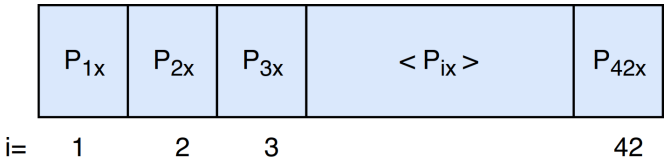

Fig. 3. Representing a valid allocation and scheduling by GA/MPGA chromosome type.

set based on a certain crossover rate. Then two genome strings exchange parts of their corresponding chromosomes to create two new genomes. In our use case, the chosen scheduling are exchanged with the other scheduling for producing two new schedules with most likely better schedules. Fig. 3 illustrates the representation of the all jobs scheduling by a genome type. Each genome consists of 42 portions since the use case has 42 different tasks. All possible assignments to processing units for each task is equal to 23 (γ). This representation also indicate the task scheduling by prioritizing the assigned tasks to the same processor. Such that the processor operates on the tasks from left to right i.e, if Task #4 (T4), Task #7 (T7)and Task #11 (T11) are assigned to Processor #2 (P2), the processor first runs T4, then T7 and T11 respectively.

Step 5. **Mutation Operator.** The main goal of mutation operator is to increase genetic diversity. Mutation alters one gene value (assigned processor to task) in a chromosome string from its initial state. The solution may be better or even worst solution by using mutation. Mutation forces GA to get rid of local optima. For doing mutation, we need to randomly select one gene in chromosome and modify its assigned value to a new valid number.

After each cycle of selection, crossover and mutation, the newly generated set of solutions (schedules) is called as new generation. All the generations are evaluated based on the fitness function to determine if they represent a good enough solution to satisfy the fitness function. This determines if the GA can stop searching, or if otherwise, for the GA to continue searching until the predefined stopping criteria is met. The stopping criteria could be the number of generations, or evolution time, or fitness threshold, or fitness convergence, or population convergence. In our case, the number of generations was set as the stopping criteria. The schedule obtained after the stopping criteria will be the optimal or near optimal schedule.

A. MPGA Algorithm

Although EC methods can improve the quality of results, using them have some difficulties. First of all, an evolutionary algorithm may not converge towards the optimal solutions or even to near-optimal solutions in the case of very huge exploration space. One possible solution is to increase the initial population size, but leading to increase the execution time of evolutionary algorithms. Parallelizing these algorithms can remarkably diminish their execution time and improve the quality of results. In the parallelized GA, multiple processors

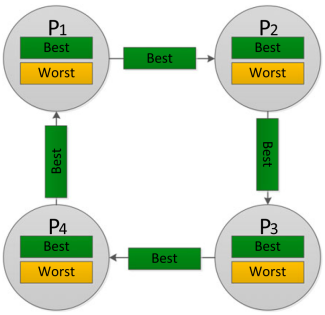

Fig. 4. Multi-population migration operation.

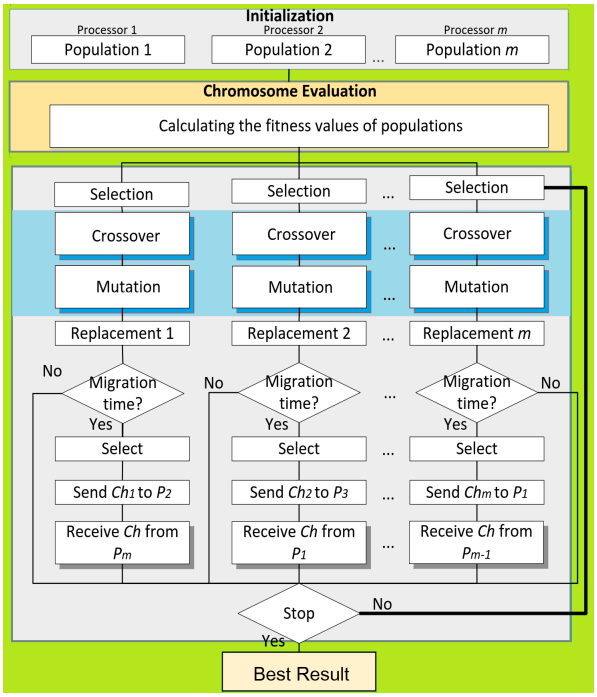

Fig. 5. Flowchart of MPGA.

work together where each one runs a simple GA and has an independent populations.

Step 6. After a predefined number of iterations, all processors share their best chromosomes among each other (***migration operation***).

Step 6 above is specific to the parallel procedure, which, including the previous 5 steps is called MPGA. Sharing the best individuals aids the MPGA to get avoid of local optima. This procedure comes to be utilized in the algorithm that we propose in further.

Fig. 4 represents the behavior of the MPGA and the flowchart of consequent operations is shown in Fig. 5. The pseudo-code of MPGA is presented in Algorithm 1. The inputs of proposed meta-heuristic optimization approach include: ① the specification of processing units including maximum processing potential, the total number of input/output ports, and ② the specifications of jobs and tasks including load complexity, run-time deadlines, and task dependencies.

Algorithm 1: Pseudo Code of MPGA

Input: • Processor P_i: $1 \leq i \leq$ # Processors
• Distributed processing units Specifications
• Jobs and related tasks Specifications
• N: Population Size
• T: Maximum Number of Iterations
Output: A Set of Near-Optimal Solutions
Function MPGA(N, T)**:**
  (***Step 1***)**:** $U_{i,0}$**= Random_Population** (N); //Creating initial random population and assign to each P_i
  (***Step 2***)**: Fitness_Function** ($U_{i,0}$); //Evaluating the objectives of each solution in the all populations
  $t = 1$;
  while $t \leq T \mid SatisfyingUserNeeds$ **do**
    (***Step 3***)**:** $U'_{i,t}$ **= Select** ($U_{i,t}$); //Select some chromosomes from the $U_{i,t}$ randomly.
    (***Step 4***)**:** $U''_{i,t}$ **= Crossover** ($U'_{i,t}$)
    (***Step 5***)**:** $Y_{i,t+1}$ **= Mutation** ($U''_{i,t}$)
    (***Step 2***)**: Fitness_Function** ($Y_{i,t+1}$);
    (***Step 6***)**: if** $\#Iterations\%MigrationGap == 0$ **then**
      Select the best chromosome from $Y_{i,t+1}$
      Send the best chromosome to P_{i+1}
      Receive the best chromosome from P_{i-1}
    $t = t + 1$;
return $Y_{i,t+1}$

IV. Evaluations

This section presents the results of experiments that have been fulfilled to evaluate the impact of the proposed MPGA on the use case. The evaluations have been done based on two different optimization scenarios including single objective and multi-objective optimization. *It is necessary to mention that the single objective optimization has been solved with simple GA, while we leveraged MPGA to solve the multi-objective optimization.*

A. Implementation Details

MPGA is implemented in C++ and MPI library has been utilized for parallelization. Ring topology is used for connections between processors for running MPGA. For the implementations, an Intel Core i7-4770 CPU 3.40 GHz with 16.0 GB RAM running on 64-bit Windows 10 has been used. Seven cores have been leveraged in the parallel implementation. The specification of MPGA parameters is shown in Table I.

B. Experimental Results Convergence

One of the main limitations of evolutionary algorithms is decreasing the convergence speed by increasing the number of iterations leading to make non-convergent results in low iterations for difficult problems. Fig. 6 and Fig. 7 represent the convergence of fitness functions for both single and multi-objective optimization, respectively. It can be easily observed from the convergence figures that both strategies are highly convergent toward the improved results by contentious reduction in fitness functions as the system cost (see Equation (1) and Equation (2)).

TABLE I
MPGA Algorithm Parameters.

Parameter	Value
N: Initial Population Size (Each Processor)	100
# Populations	15
Maximum # Iterations	750
Crossover Rate	{0.1, 0.5, 0.9} per each 5 populations
Mutation	One-Point Mutation
Migration Rate	3
Migration Gap	25
Mutation Rate	1 - Crossover Rate

TABLE II
Experimental Results Compared to [11]

Exploration Approach	End-to-end finishing time	# Processing units
Our Single Objective Equation(1) (Solved by simple GA)	250	7
Our Multi-Objective Equation(2) (Solved by MPGA)	Solution①: 210 Solution②: 250 Solution③: 160	7 8 9
Single Objective [11]	210	9
Multi-Objective [11]	180	11

a) Single Objective Optimization. Fig. 8 illustrates the variation trend of total number of utilized processing units over the number of iterations. As mentioned before, the aim of single objective optimization is to decrease the number of processing units used in jobs scheduling. Fig. 8 shows considerable improvement in finding scheduling with less required processing units. According the results of Table II, we need 22 processing units for scheduling in the first iteration, while by proceeding the exploration algorithm, we found a solution with only seven required processing units. Although there exist some breaks in continuous improvement, the overall trend moves toward improvement.

b) Multi-Objective Optimization. As mentioned before, the total number of processing units and end-to-end finishing time (represented as $Run-time$ in (2)) for all the jobs are the two main objectives of MPGA. Fig. 9 and Fig. 10 illustrate the convergence figures of required processing units for scheduling and end-to-end finishing time for all the scheduled jobs, respectively. We can conclude from the figures that both the objectives are approaching toward optimized results. Although there are some failures or stops in achieving better results in each iteration, the overall Progression of MPGA always approaches toward superior outcomes (Fig. 11).

Table II shows three different solutions on the Pareto frontier of the last Population. We have a variety of options based on the user needs. Solution① is an scheduling with minimized number of processing units (7 processing units) while takes more time, 210tu, for running. On the other hand,

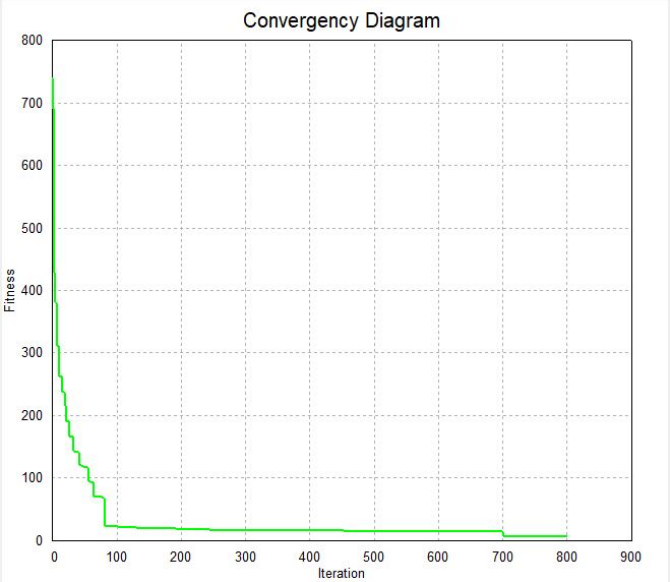

Fig. 6. Convergence diagram of the fitness function for the single objective optimization ((1)).

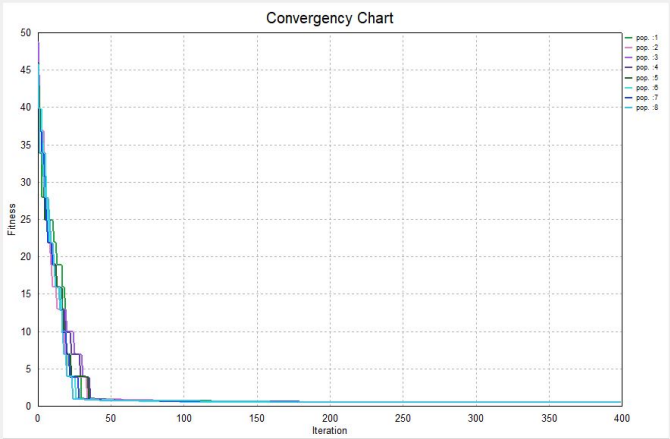

Fig. 7. MPGA convergence diagram of the fitness function for eight different populations ((2)).

Solution③ provide the minimum elapsed end-to-end finishing time (160tu), while needs 9 processing units for running.

C. Comparison between MPGA and simple GA

For evaluating the impact of multi-population optimization on the allocation and scheduling problem, the results of single objective optimization has been achieved by leveraging single population GA (simple GA). On the other hand, the results of multi-objective optimization has been achieved by using MPGA. We compared MPGA and simple GA schemes in terms of exploration time and quality of results in the following sections.

a) Exploration Time and Speedup. Fig. 8 and Fig. 9 represent the convergence of processing units for single population GA and MPGA, respectively. MPGA achieve the best result after 400 iterations, while single population GA needs 750

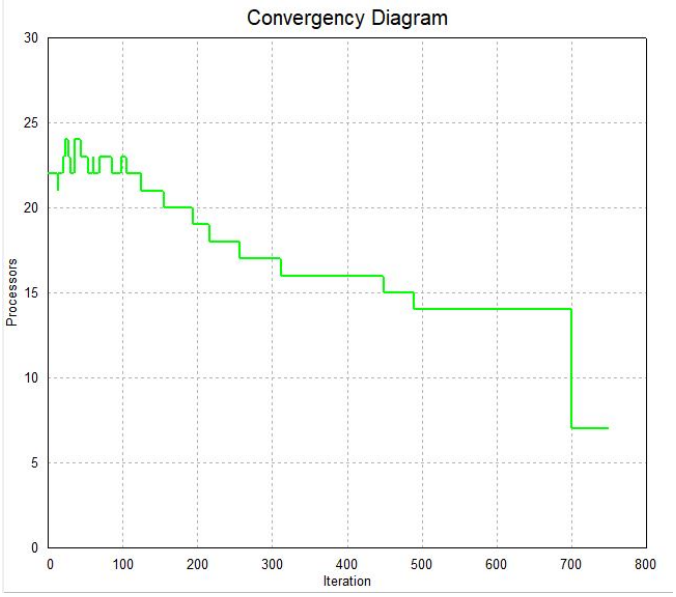

Fig. 8. Convergence diagram of the variations # processing units in single objective optimization solved by simple GA.

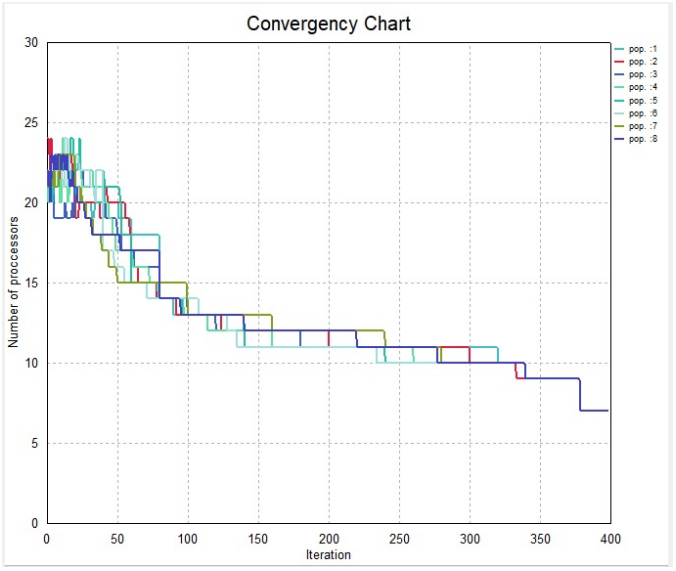

Fig. 9. Convergence diagram of # processing units in multi-objective optimization by using MPGA approach.

iterations for finding the best solution. Obviously converging to the best result needs in less number of iterations by using MPGA which is the best proof to show the benefits of applying the MPGA, especially when the design space is large.

b) Quality of Results. According to the results of Table II, MPGA found a solution with 7 required processing units and 210tu for the end-to-end finishing time, while single population GA found a solution with the same required processing units but takes 250tu for the end-to-end finishing time. MPGA provides more quality of results compared to single population GA even when single population GA tries to optimize only one

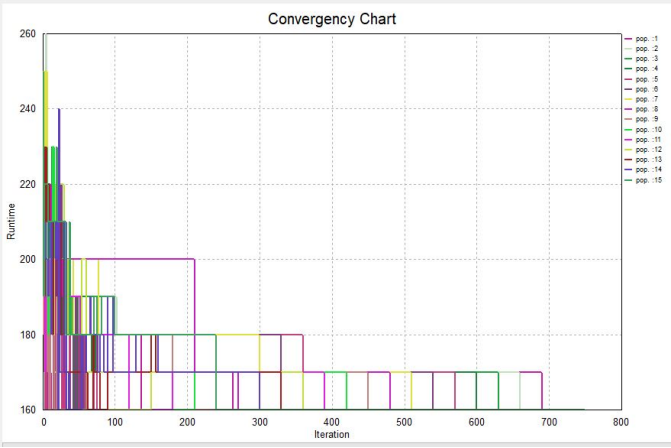

Fig. 10. Convergence diagram of end-to-end run-time in multi-objective optimization by using MPGA approach.

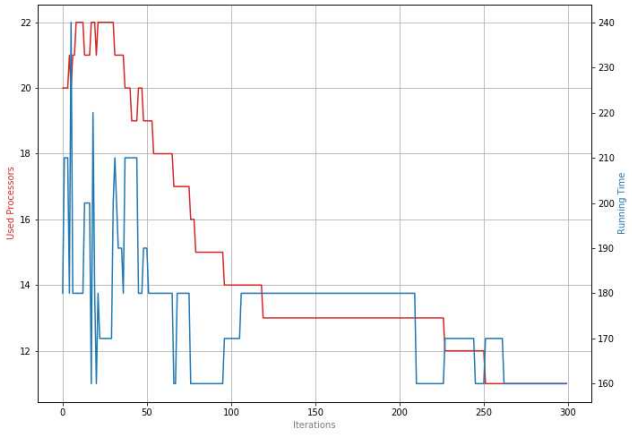

Fig. 11. Improvement proceeding of exploration objectives including the number of processing units and end-to-end use case run-time.

objective.

D. Comparison Between MPGA and Morady et al. [11]

We compare the proposed MPGA solution with a similar evolutionary approach [11]. However, we customize the fitness functions for the studied use case. Table II represents the evaluation results after applying [11] on the industrial use case. As seen in Table II, our method in single objective scenario found a scheduling with 7 required processing unit, while [11] proposed a solution with 9 required processing units. In addition, in multi-objective scenario, [11] needs 180tu to finishe all the jobs, while compared to Solution③, our proposed method needs 160tu. Therefore we can conclude our customized MPGA overcomes other similar EC methods.

E. Allocation and Scheduling Results

We have considered here the use case described in section II, and illustrated entirely by Fig. 1. After applying MPGA to the use case tasks, the near-optimal scheduling result is shown in Fig. 12, a valid scheduling for all jobs and their related tasks with minimum number of processing units (Solution①).

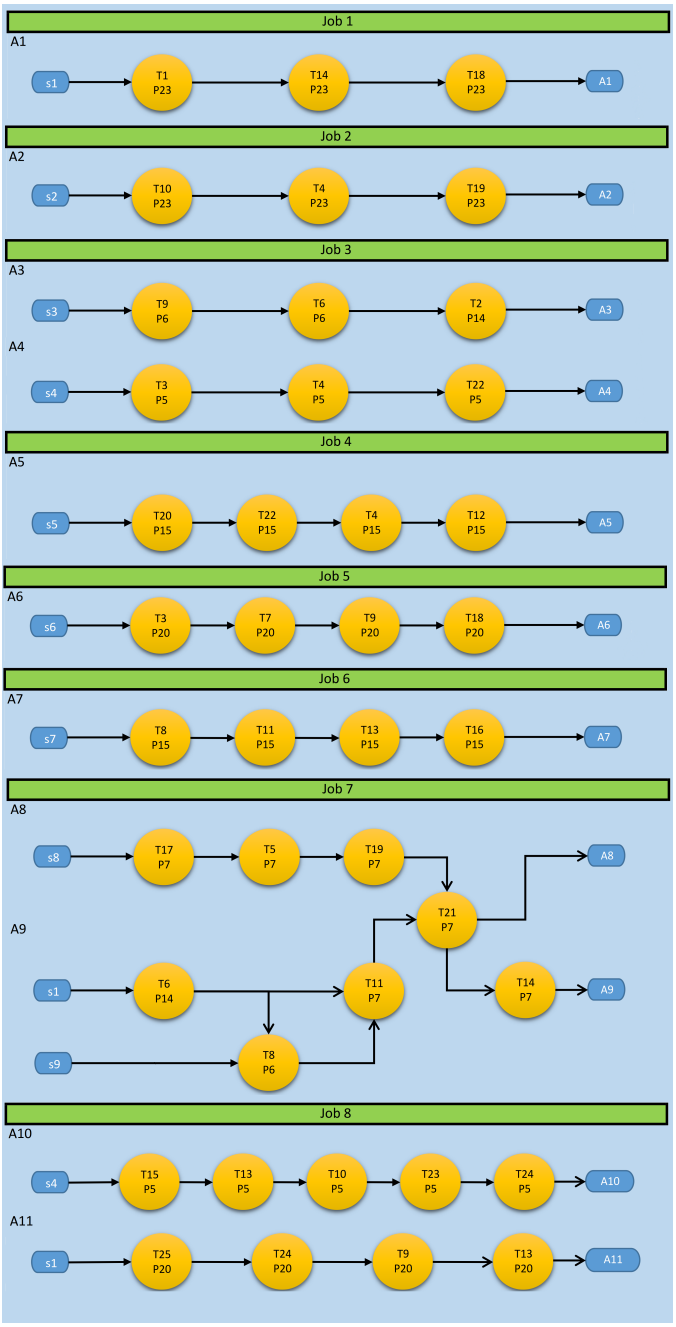

Fig. 12. The best solution for multi-objective optimization with minimum number of processing units (solution①)

V. Related Work

Here, we first explore more traditional list scheduling heuristics that have considered communication costs.

The basic idea is to make an ordered list of nodes by assigning them orders, and then to repeatedly execute the following two steps until a valid schedule is obtained: ① Select from the list the node with the highest order for

scheduling. ② Select a processor to accommodate this node. In realistic cases, scheduling needs to exploit parallelism by identifying the task graph structure and take into consideration task granularity, arbitrary computation, and communication costs.

In [10], the modified critical path algorithm (MCP) is proposed, based on the latest possible start time of a node. A node's latest possible start time is determined via the as-late-as-possible (ALAP) binding by traversing the task graph upward from the exit nodes to the entry nodes while pulling the node's start times downwards as much as possible. The latest possible start time of the node itself is followed by a decreasing order of the latest possible start times of its successor nodes. Furthermore, in [10], the dominant sequence clustering algorithm (DSC) is presented. It is based on the dominant sequence, which is essentially the critical path of the partially scheduled task. CP (the critical path of task graph) node is a ready node. If so, DSC schedules it to a processor allowing the minimum start time. Such a minimum start time may be achieved by rescheduling some of the node's predecessors to the same processor. If the highest CP node is not a ready node, DSC does not select it for scheduling. Instead, it chooses the highest node which lies on a path reaching the CP for scheduling. Moreover, also in [10], the mobility directed algorithm (MD) is presented. MD selects a node at each step based on relative mobility which is defined as the difference between a node's earliest start time and latest start time. Similar to the ALAP binding, the earliest possible start time is assigned to each node via the as-soon-as-possible (ASAP) binding. This is performed by traversing the task graph downward from the entry nodes to the exit nodes while pulling the nodes upward as much as possible. Moreover, relative mobility is obtained by dividing the mobility with the nodes computation cost. Basically, a node with zero mobility is a node on the CP. At each step, MD schedules the node with the smallest mobility to the first processor having a large enough time to accommodate the node without considering the minimization of the nodes start time. After a node has been scheduled, the relative mobility values of the remaining nodes are updated.

In [11], a MPGA is presented which outperforms deterministic and non-deterministic methods described in [12], [13]. In [14], a new encoding mechanism with a multi-functional chromosome is presented, using a priority representation that is called priority- based multi-chromosome (PMC). PMC can efficiently represent a task schedule and assign tasks to processors. It is another meta-heuristic method that uses a GA to achieve near-optimal scheduling of tasks.

Research on static mapping methods includes the work of Lei et al., who proposed a genetic mapping algorithm to optimize application execution time [15]. In their work, graphs represent applications and the target architecture is a NoC. Wu, et al. also investigated genetic mapping algorithms [16]. By combining dynamic voltage scaling techniques with mapping, they achieved 51% savings in energy consumption. Murali et al. explored mappings for more than one application in NoC design, using the tabu search (TS) algorithm [17]. Manolache, et al. investigated task mapping in NoCs, trying to guarantee packet latency [18]. For this purpose, both the task-mapping algorithm (TS) and the routing algorithm are defined at design time. Hu et al. presented a branch-and-bound algorithm to map a set of IP cores (IPs) onto a NoC with bandwidth reservation [19]. Their results show energy savings of 51.7% in the communication architecture. In [20] presented a task scheduling scheme on heterogeneous computing systems using a multiple priority queues genetic algorithm (MPQGA). Their experimental results for large-sized problems for a large set of randomly generated graphs as well as graphs of real-world problems with various characteristics showed that the proposed MPQGA algorithm outperformed two non-evolutionary heuristics and a random search method.

VI. Conclusions and Future Work

Leveraging a distributed environment for task scheduling can enhance reliably and provide a specification compliant processing scheme. The inherent difficulties in distributing application jobs and scheduling them among processing units may lead applications to expose low performance, or the system may require extra (unnecessary) resource costs. Here, a parallel Multi-Population Genetic Algorithm is developed to overcome complexity barriers, towards optimizing both operation time and resource numbers, while preserving application requirements. For the evaluations, a synthetic use case has been studied, grouping many aspects of actual industrial systems. The final results offer a better resource efficiency (requiring less number of processing unit) while guaranteeing real-time execution. In addition, MPGA provides better efficiency compared to other cutting-edge evolutionary approaches. We expect more complex problems to appear when we need to deal with duplication of tasks and synchronization activities, related to reliability and fault tolerance aspects, in future research actions.

References

[1] Freund, R. F. Optimal selection theory for superconcurrency. Proc. Supercomputing 89. IEEE Computer Society, Reno, NV, 1989, pp. 699703.

[2] Adam TL, Chandy KM, Dicksoni JR. A comparison of list schedules for parallel processing systems. Communications of the ACM 1974;17(12):68590.

[3] Wu MY, Gajski DD. Hypertool: a programming aid for message-passing systems. IEEE Transactions on Parallel and Distributed Systems 1990;1(3):33043.

[4] Hou ESH, Ansari N, Hong R. A genetic algorithm for multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems 1994;5(2):11320.

[5] Hwang RK, Gen M. Multiprocessor scheduling using genetic algorithm with priority-based coding. Proceedings of IEEE conference on electronics, information and systems; 2004.

[6] Wu AS, Yu H, Jin S, Lin K-C, Schiavone G. An incremental genetic algorithm approach to multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems 2004;15(9):82434.

[7] Majd, Amin, et al. "NOMeS: Near-optimal meta-heuristic scheduling for MPSoCs." Computer Architecture and Digital Systems (CADS), 2017 19th International Symposium on. IEEE, 2017.

[8] Majd, Amin, Golnaz Sahebi, Masoud Daneshtalab, Juha Plosila, Shahriar Lotfi, and Hannu Tenhunen. "Parallel imperialist competitive algorithms." Concurrency and Computation: Practice and Experience 30, no. 7 (2018): e4393.

[9] Y. Chen, Y. Zhong, Automatic Path-oriented Test Data Generation Using a Multi-population Genetic, Proc. Fourth International Conference on Natural Computation, pp. 566 570, Oct 2008.

[10] Adam TL, Chandy KM, Dicksoni JR. A comparison of list schedules for parallel processing systems. Communications of the ACM 1974;17(12):68590.

[11] R. Morady and D. Dal, A Multi-Population Based Parallel Genetic Algorithm for Multiprocessor Task Scheduling with Communication Costs, 2016 IEEE Symposium on Computers and Communication (ISCC).

[12] Wu MY, Gajski DD. Hypertool: a programming aid for message-passing systems. IEEE Transactions on Parallel and Distributed Systems 1990;1(3):33043.

[13] Hou ESH, Ansari N, Hong R. A genetic algorithm for multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems 1994;5(2):11320.

[14] R. Hwang, M. Gen and H. Katayama, A comparison of multiprocessor task scheduling algorithms with communication costs Computers & Operations Research, Vol. 35, pp. 976 993, ELSEVIER, 2008.

[15] T. Lei and S. Kumar, A Two-Step Genetic Algorithm for Mapping Task Graphs to a Network on Chip Architecture, Proc. Euromicro Symp. Digital System Design (DSD 03), IEEE Press, 2003, pp. 180-187.

[16] D. Wu, B. Al-Hashimi, and P. Eles, Scheduling and Mapping of Conditional Task Graphs for the Synthesis of Low Power Embedded Systems, Proc. Design, Automation and Test in Europe (DATE 03), IEEE CS Press, 2003, pp. 90-95.

[17] S. Murali and G. De Micheli, Bandwidth-Constrained Mapping of Cores onto NoC Architectures, Proc. Design, Automation and Test in Europe (DATE 04), IEEE CS Press, 2004, pp. 896-901.

[18] S. Manolache, P. Eles, and Z. Peng, Fault and Energy-Aware Communication Mapping with Guaranteed Latency for Applications Implemented on NoC, Proc. 42nd Annual Design Automation Conf. (DAC 05), ACM Press, 2005, pp. 266-269.

[19] J. Hu and R. Marculescu, Energy- and Performance- Aware Mapping for Regular NoC Architectures, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 24, no. 4, 2005, pp. 551-562.

[20] Y. Xu, K. Li, J. Hu and K. li, A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues, Information Sciences, Vol.270, pp. 255-287,Elsevier,2014.