Research Article

Journal of Algorithms & Computational Technology

Journal of Algorithms & Computational Technology
2017, Vol. 11(1) 58–67

DOI: 10.1177/1748301816665521
journals.sagepub.com/home/act
SAGE

Intelligent discrete particle swarm optimization for multiprocessor task scheduling problem

S Sarathambekai and K Umamaheswari

Abstract

Discrete particle swarm optimization is one of the most recently developed population-based meta-heuristic optimization algorithm in swarm intelligence that can be used in any discrete optimization problems. This article presents a discrete particle swarm optimization algorithm to efficiently schedule the tasks in the heterogeneous multiprocessor systems. All the optimization algorithms share a common algorithmic step, namely population initialization. It plays a significant role because it can affect the convergence speed and also the quality of the final solution. The random initialization is the most commonly used method in majority of the evolutionary algorithms to generate solutions in the initial population. The initial good quality solutions can facilitate the algorithm to locate the optimal solution or else it may prevent the algorithm from finding the optimal solution. Intelligence should be incorporated to generate the initial population in order to avoid the premature convergence. This article presents a discrete particle swarm optimization algorithm, which incorporates opposition-based technique to generate initial population and greedy algorithm to balance the load of the processors. Make span, flow time, and reliability cost are three different measures used to evaluate the efficiency of the proposed discrete particle swarm optimization algorithm for scheduling independent tasks in distributed systems. Computational simulations are done based on a set of benchmark instances to assess the performance of the proposed algorithm.

Keywords

Distributed systems, particle swarm optimization, scheduling, swarm intelligence

Date received: 8 September 2015; accepted: 31 March 2016

Introduction

The fast growth of parallel and distributed computing environments, driven by increasing demand for computing power, encourage a variety of distributed computing platforms emerging in academic and industrial communities, such as grid and cloud computing.[1] A challenging issue in distributed systems is task scheduling (TS). TS is a complex combinatorial optimization problem because the search space increases exponentially with the problem size. The traditional or conventional methods for TS need more time to locate the optimum solution. To get a near optimal solution within a finite duration, heuristics are used instead of exact optimization methods. Braun et al.[2] presented a comparative study on 11 heuristics for the TS problem and evaluated them on different types of heterogeneous environments. These heuristics are intended to minimize only a single objective, the make span of the schedule. Izakian et al.[3] compared six efficient heuristics such as min-min, max-min, min-max, LJFR-SJFR, suffrage, and work queue for scheduling meta-tasks on heterogeneous distributed environment. The effectiveness of these heuristics are estimated by make span and flow time.

To improve the quality of solutions, meta-heuristics have been presented for TS problem in distributed systems. The most popular meta-heuristic algorithms in the literature are Genetic Algorithm (GA),[4]

Department of IT, PSG College of Technology, Coimbatore, India

Corresponding author:
S Sarathambekai, Department of IT, PSG College of Technology, Coimbatore, Tamil Nadu, India.
Email: vrs070708@gmail.com

Differential Evolution (DE),[5] and Particle Swarm Optimization (PSO).[6]

Discrete particle swarm optimization (DPSO) is a recently developed population-based meta-heuristic algorithm proposed by Kang and He.[7] The performance of the DPSO was tested using a single criterion such as makespan. In this algorithm, the particles are encoded in position vector representation. This encoded scheme doesn't convey the flow of execution of tasks in the processors. The flow is not required, if the algorithm optimizes only the makespan. Because, the make span value cannot be changed, if the order of the tasks assigned within a processor varies. The single criterion is not enough to test the performance of the DPSO algorithm. This problem is addressed by Sarathambekai and Umamaheswari.[8] In Sarathambekai and Umamaheswari,[8] the multiple objectives such as makespan, flow time, and reliability cost are used to evaluate the efficiency of the multi-objective discrete particle swarm optimization (MDPSO) algorithm. Here, the particles are represented in permutation-based method because the metric flow time depends on the order of execution of the tasks in the processors. This article presents the MDPSO algorithm for scheduling of independent tasks in distributed systems.

Most of the existing works in the literature are focused on introducing or improving the control parameters and adaptive controlling of parameter settings in PSO. Population initialization can also affect the convergence speed and quality of the final solution. There is only limited research in this field. Kazimipour et al.[9] investigated the effect of population initialization methods for Evolutionary Algorithms (EA) on Large Scale Global Optimization (LSGO) problems. The well-known existing initialization methods are categorized into five major categories such as Stochastic, Deterministic, Two-step, Hybrid, and Application specific methods. Kazimipour et al.[9] suggested that the random initialization is not a proper choice for population initialization even with very large population size and also recommended to use two step methods to improve the performance of the EA. The two-step methods generate an initial population in the first step and then try to improve them using fitness function in the second step.

The Opposition-Based Learning (OBL) initialization is a two-step method proposed by Tizhoosh.[10] This has been included in some machine learning algorithms like opposition-based reinforcement learning,[11] opposition-based DE,[12] and opposition-based GA.[13] Wang and Liu[14] proposed opposition-based PSO. This method employs OBL and dynamic Cauchy-based mutation to avoid premature convergence in classical PSO. Omran[15] proposed two variants such as improved PSO (iPSO) and improved DE (iDE) based on OBL. The iPSO and iDE algorithms replace the least-fit solution with its opposite solution in each iteration. Wang and Liu[14] stated that OBL enhances the performance of PSO and DE without adding any additional parameters. All these OBL-based initialization are for solving continuous domain optimization problems. Ergezer[16] proposed OBL for discrete problems and presented two different methods of OBL such as Open Path Opposition (OPO) and Circular Opposition (CO) in solving discrete and combinatorial optimization problems. The effectiveness of the opposition techniques[16] are simulated on traveling salesman problem using biogeography-based optimization algorithm. The OPO is suitable for combinatorial problems where the final node is not connected to the first node. In CO technique, the final node of a graph is linked to the first node. The proposed research work has taken OPO[16] in swarm (population) initialization of DPSO, because here the tasks are independent; there are no links between the tasks.

The load balancing is the central component of distributed systems. The purpose of load balancing is to improve the performance of a distributed system through an appropriate distribution of the application load. This is one of the important issues when the demand for computing power increases in the distributed computing environments. Load balancing algorithms can be broadly classified into static and dynamic. The main benefit of static load balancing is that all the overhead of the scheduling process is incurred at compile time, resulting in a more efficient execution time environment compared with dynamic scheduling methods. The proposed work uses a static load balancing algorithm to distribute the task among processors prior to the execution of the algorithm in order to avoid the overhead of the scheduling process during the executing time.

The remainder of the article is organized as follows: The next section describes the problem formulation and the following section presents the proposed DPSO. Experimental results are reported in "Simulation results and analysis" section and the final section concludes the article.

Problem formulation

Task model

A Heterogeneous Computing (HC) system consists of a number of heterogeneous Processor Elements (PEs) connected with a mesh topology. Let $T = \{T_1, T_2 \ldots, T_n\}$ denote the n number of tasks that are independent of each other to be scheduled on m processors $P = \{P_1, P_2 \ldots, P_m\}$. Because of the heterogeneous nature of the processors and disparate nature of the tasks, the expected execution times of a task executing on

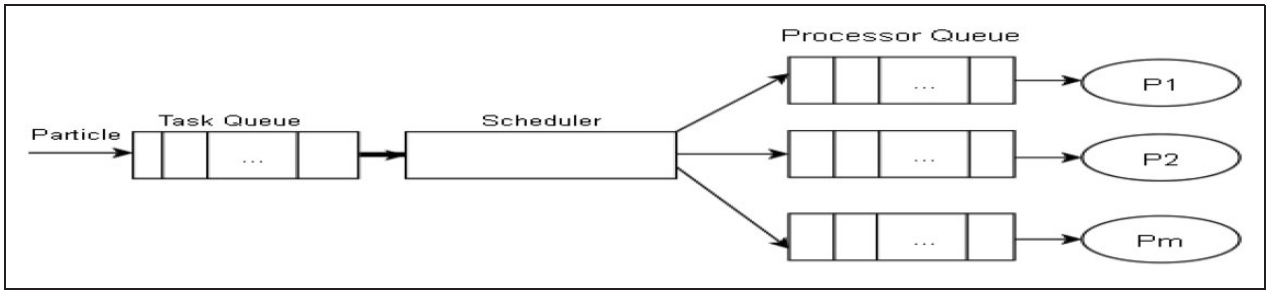

Figure 1. Scheduling model for heterogeneous environment.

different processors are different. Every task has an Expected Time to Compute (ETC) on a specific processor. The ETC values are assumed to be known in advance. An ETC matrix is $n \times m$ matrix in which m is the number of processors and n is the number of tasks. One row of the ETC matrix represents estimated execution time for a specified task on each PE. Similarly, one column of the ETC matrix consists of the estimated execution time of a specified PE for each task.

The TS problem is formulated based on the following assumptions:

- All tasks are available at zero time.
- Processors are always available.
- The execution time of each task on each processor is known and constant.
- Preemption is not allowed.
- Each processor can process only one task at a time.
- A task cannot be processed on more than one processor at a time.
- Each processor uses the First-Come, First-Served (FCFS) method for performing the received tasks.

Scheduler model

A static scheduler model in distributed systems is shown in Figure 1. The scheduler manages two queues such as Task Queue (TQ) and Processor Queue (PQ). The scheduling algorithm in the central scheduler is started to work with TQ. TQ contains a set of tasks in a particle. The scheduler is responsible for distributing each task in the TQ to the individual PQ based on the workload of the each processor in the distributed systems. Once the scheduler completes to place all the tasks from the TQ to PQ, the processors will start executing the tasks in their own PQ.

Proposed DPSO

The previous research work[8] presented DPSO approach for scheduling problem. This DPSO algorithm performs random initialization. Population initialization plays an important role because it can affect the convergence speed and also the quality of the final solution. To enhance the quality of the solution, this article proposed DPSO algorithm namely Intelligent DPSO (IDPSO), which mainly focuses to fine tune the population initialization with the help of OPO technique and Greedy Load balancing algorithm. Also, the algorithm schedules independent tasks in the distributed systems with subject to the load of each processor.

The following subsections describe in detail the steps of IDPSO algorithm.

Swarm initialization

Swarm initialization consists of two parts: Particle initialization and Processor allocation. Number of tasks (n) and population size (N) are required to generate particles. Here, a particle is encoded in permutation-based method. In the permutation vector, the position of a task represents the sequence the task is scheduled and the corresponding value indicates a task number.

***Pseudo code* 1**: Open-path greedy opposite algorithm

Input: Swarm with size N; Output: Swarm with size 2N

for each particle in a swarm **do**
  for each task in a particle Task_i **do**
    // i varies from 1 to n
    if Task_i is odd **then**
      $\text{Task}_{\text{Opposite}} \leftarrow \frac{\text{Task}_i+1}{2}$
      // $\text{Task}_{opposite}$ is the opponent of Task_i
    else // Task_i is even
      $\text{Task}_{\text{Opposite}} \leftarrow n + 1 - \frac{\text{Task}_i}{2}$
    end if
  end for
end for

Initially, particle initialization generates random particles called original swarm and then the opponent

particles of the original swarm are calculated using OPO technique,[16] which is shown in pseudo code 1. Now, both swarms are merged and the best particles according to fitness function are selected to form the initial swarm for the IDPSO.

Random processor allocation is a simple and straightforward technique. However, the problem in this technique is, some processors are busy with processing, while some processors are idle without any processing. To make better utilization of the processors, the IDPSO performs load balancing using Heuristic-based Greedy static Load balancing algorithm (HGL), which is shown in pseudo code 2.

***Pseudo code* 2**: Greedy Load balancing algorithm

```
Input: Particle, ETC
    for each task Task_i in a TQ do
      // i varies from 1 to n; n is the number of tasks
      if (m>n) then            // m is the number
        of processors
         Place Task_i in TQ to the fittest available
           processor's PQ based on the ETC value.
      else
        for each Processor_j in the processor
            list do            // j varies from 1 to m
          Select the least loaded Processor_j based
              on the load of the PQ
        end for
          Place Task_i to the Processor_j's PQ.
          Update load of the Processor_j that
            receives Task_i in the PQ
      end if
    end for
```

The resource utilization[17] is the performance criterion for the scheduler to perform scheduling with balancing the load. The processor's utilization is defined as the percentage of time that processor P_j is busy during the scheduling time. The processor's utilization PU_j for the processor P_j is calculated using equation (1).

$$\mathrm{PU_j} = {}^{\mathrm{Aval(P_j)}}\!/_{\mathrm{Makespan}} \quad \text{for } \mathrm{j} = 1 \ldots \mathrm{m} \tag{1}$$

where $Aval(P_j)$ is the processor's availability time. The processor availability time is the time when the processor P_j completes the execution of all the assigned tasks. Make span[8] (finishing time of the newest task) computes the throughput of the HC system.

The resource utilization RU is the average of processor's utilization, which is calculated using equation (2).

$$\mathrm{RU} = \frac{\sum_{\mathrm{j}=1}^{\mathrm{m}} \mathrm{PU_j}}{\mathrm{m}} \tag{2}$$

Figure 2 shows an illustrative example for a swarm initialization, which corresponds to the scheduling of five independent tasks that assigns to the three heterogeneous processors using random allocation and allocation with balancing the load using HGL. The processors utilization is 70% in random allocation and 94% in HGL. Hence, the HGL-based allocation utilizes the resources efficiently.

Particle evaluation

The previous research work[8] presented DPSO approach for scheduling problem to minimize make span, flow time, and reliability cost. The make span and flow time values are in incomparable ranges and the flow time has a higher magnitude order over the make span. To address this problem, the proposed IDPSO algorithm is assessed by using the three evaluation criteria such as make span,[8] mean flow time instead of flow time and reliability cost.[8]

The value of mean flow time[18] is used to evaluate flow time. Assume k is the total number of tasks assigned to processor P_i and F_{ji} is the finishing time of *task* T_j on a processor P_i. The calculation of mean flow time is given in equation (3).

$$\text{Mean Flow time} = \frac{\sum_{\mathrm{i}=1}^{\mathrm{m}} \mathrm{M\_Flow_i}}{\mathrm{m}} \tag{3}$$

$$\mathrm{M\_Flow_i} = \frac{\sum_{\mathrm{j}=1}^{\mathrm{k}} \mathrm{F_{ji}}}{\mathrm{k}} \tag{4}$$

The three objectives are used to evaluate the performance of the scheduler. The weighted single objective function called Adaptive Weighted Sum[19] is used to calculate the fitness value of each particle in the swarm. This can be estimated using equation (5), where $\alpha_i 1 \varepsilon [0, 1]$.

$$\begin{aligned} &\text{Fitness} = \alpha_1\alpha_2\text{Makespan} + (1-\alpha_1) \\ &\alpha_2\text{Mean Flow time} + (1-\alpha_2)\text{Reliability Cost} \end{aligned} \tag{5}$$

Update the particle's Personal best (Pbest) and Global best (Gbest) position

The Pbest position of each particle and Gbest position of the swarm can be determined based on the fitness

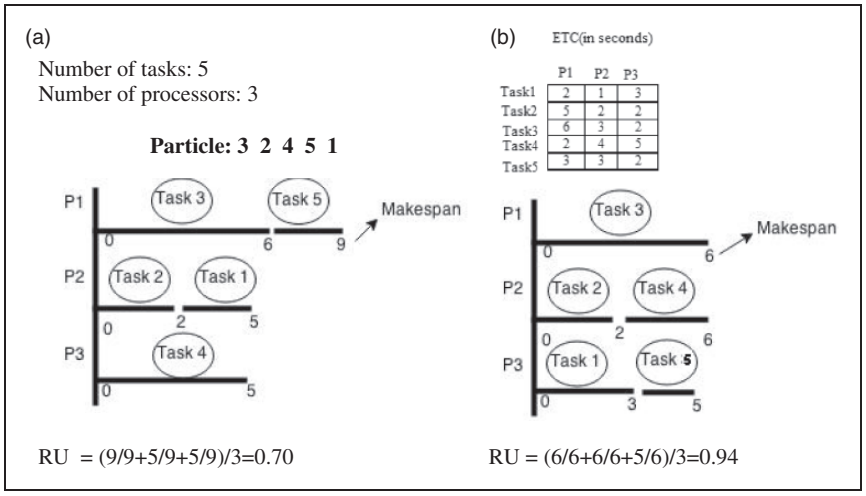

Figure 2. Processor allocation. (a) Random Processor Allocation (b) Processor Allocation with load balancing.

value. The Pbest and the Gbest should be determined before updating the position of the particle.

Updating the particle's velocity and position

The particles in IDPSO update their velocity and position using equations (6) and (10), respectively.[8]

$$V_i^{(t+1)}(j) = WV_i^t(j) + C_1 r_1\left(Pbest_i^t(j) - present_i^t(j)\right) + C_2(1 - r_1)(Gbest^t(j) - present_i^t(j)) \quad (6)$$

The inertia weight (W) in equation (6) is calculated using equation (7).

$$W = W_0\left(1 - \frac{H_i}{MDH}\right) \quad (7)$$

$$H_i = \text{Hamming distance}(Gbest^t, present_i^t) \quad (8)$$

$$MDH = Max(H_i) \quad (9)$$

$$present_i^{(t+1)}(j) = present_i^t(j) + V_i^{(t+1)}(j) \quad (10)$$

where $i = 1, 2, 3\ldots N$; $j = 1, 2, 3\ldots n$; N is the swarm size and n is the size of particle. W is the inertia weight which is used to control the impact of the previous history of velocities V_i^t on the current velocity of a given particle. Linearly Decreasing Inertia (LDI) weight was used in the existing DPSO[7,8] algorithm. This kind of the inertia factor does not guarantee that the particles have not moved away from the Gbest. Instead of LDI weight, the IDPSO uses hamming distance based inertia. Moving the particles based on the hamming distance between the particles and their Gbest that ensures that the particles have not moved away from the Gbest. The Hamming inertia weight is calculated using equation (7), where W_0 is the random number between 0.5 and 1, H_i is the current hamming distance of i^{th} particle from the Gbest, and MDH is the maximum distance of a particle from the Gbest in that generation. $V_i^{(t+1)}(j)$ is the j^{th} element of the velocity vector of the i^{th} particle in $(t+1)^{th}$ iteration, which determines the direction in which a particle needs to move. The $present_i^t(j)$ is j^{th} element of i^{th} particle in t^{th} iteration. r_1 is the random value in range [0,1] sampled from a uniform distribution. C_1 and C_2 are positive constants, called acceleration coefficients, which control the influence of Pbest and Gbest on the search process.

VND heuristic

Local search technique was not included in MDPSO[8] algorithm. Therefore, the algorithm may get stuck in local optima. The IDPSO algorithm applies VND[7] (Variable NeighborhooD) local search heuristic when there is no appreciable improvement in the Gbest particle for five iterations.

Stopping condition

The above iterative processes on a swarm will continue until a pre-defined maximum number of iterations have been reached or no significant improvement in the fitness value for more than 10 iterations.

The average Relative Percentage Deviation (RPD)[7] is also calculated along with makespan, flow time, and reliability cost for comparing the results of the DPSO and IDPSO algorithms. It is calculated using equation (11), where P is the average result of the proposed algorithm and AC_i is the average result provided by other algorithms for each instance.

$$RPD = (AC_i - P)/P*100 \quad (11)$$

Simulation results and analysis

The simulation results are attained using a set of benchmark ETC instances[20] for the distributed heterogeneous systems. The algorithms are coded in Java and executed in Net Beans IDE.

Benchmark instances description

The simulation is performed on the benchmark ETC instances,[20] which are categorized in 12 types of ETC's based on three metrics: task heterogeneity, machine heterogeneity, and consistency.[7]

All instances consisting of 512 tasks and 16 processors are classified into 12 different types of ETC matrices according to the above three metrics.

The instances are labeled as g_a_bb_cc as follows:

- g means gamma distribution used in generating the matrices.
- a shows the type of consistency, c means consistent, i means inconsistent, and s means semi-consistent.
- bb indicates the heterogeneity of the tasks, hi means high, and lo means low.
- cc represents the heterogeneity of the machines, hi means high, and lo means low.

Parameter setup

The following parameters are initialized for simulating the DPSO and IDPSO algorithms.

- Number of iteration is set to 1000.
- The failure rate for each processor is uniformly distributed[21] in the range from 0.95×10^{-6}/h to 1.05×10^{-6}/h.

Performance comparisons

The improvement of DPSO[7] over the existing algorithms such as PSO, Re-excited PSO, GA, Simulated annealing, and Tabu search is 7.1%, 7.45%, 4.72%, 8.54%, and 3.35% across all ETC instances, respectively.[7] Therefore, the comparison of the IDPSO has been made only with DPSO.[7]

The population size of the DPSO algorithm is set to 32 as recommended in Kang and He.[7] The size of the population is an important factor which affects the performance of the algorithm. There is a possibility to trap the algorithm in local optima (premature convergence), if the size of the population is very less. Therefore, this research work was performed on varying size of the population. The average results of the existing DPSO algorithm with population size 32 and 50 are given in Table 1.

Table 1. The DPSO algorithm with varying population size.

ETC instance	Makespan (s)			Mean flow time (s)			Reliability cost		
	DPSO-32	DPSO-50	RPD	DPSO-32	DPSO-50	RPD	DPSO-32	DPSO-50	RPD
c_lo_lo	64146.83	**36531.98**	75.5909	**14691.31**	15562.18	−5.596	**0.453008**	0.4877706	−7.127
c_lo_hi	**36733.84**	53500.645	−31.339	15586.91	**14355.56**	8.577	0.486978	**0.447082**	8.9237
c_hi_lo	35480.39	**31716.48**	11.8674	12033.85	**11480.2**	4.822	0.376999	**0.362982**	3.8617
c_hi_hi	**33517.27**	36869.387	−9.0919	**15766.85**	16123.60	−2.213	**0.477446**	0.4923322	−3.024
i_lo_lo	**25244.15**	25372.918	−0.5075	**11116.12**	11725.77	−5.199	**0.365873**	0.3682831	−0.654
i_lo_hi	29087.27	**28745.82**	1.18782	14067.264	**13523.78**	4.018	0.443953	**0.434496**	2.1765
i_hi_lo	25326.51	**21136.15**	19.8256	11046.564	**8495.424**	30.03	0.356682	**0.290154**	22.929
i_hi_hi	**30726.37**	30867.932	−0.4586	14973.15	**14913.3**	0.401	**0.456358**	0.4584314	−0.452
s_lo_lo	26123.70	**23940.33**	9.12007	9804.628	**8712.591**	12.53	0.324445	**0.301312**	7.6776
s_lo_hi	31068.38	**30302.91**	2.52608	14389.144	**14089.5**	2.126	0.456492	**0.445036**	2.5742
s_hi_lo	27297.06	**23675.34**	15.2975	10537.472	**8848.901**	19.08	0.334838	**0.303039**	10.494
s_hi_hi	32314.79	**31235.64**	3.4549	15551.636	**15339.87**	1.380	0.475777	**0.469574**	1.3211
Average			8.12273			5.83044			4.05835

Note. The values in bold indicate that the algorithm provides better results than other algorithm.

Table 2. Comparison of objective values of IDPSO with DPSO-50.

ETC instances	Makespan (s)			Mean flow time (s)			Reliability cost		
	DPSO-50	IDPSO	RPD	DPSO-50	IDPSO	RPD	DPSO-50	IDPSO	RPD
c_lo_lo	36531.98	**13951.22**	161.85	15562.186	**7020.537**	121.667	0.48777	**0.207334**	135.25
c_lo_hi	53500.65	**30016.18**	78.239	14355.561	**14192.89**	1.14617	**0.44708**	0.464944	−3.8417
c_hi_lo	31716.48	**14041.7**	125.87	11480.204	**6771.882**	69.5275	0.36298	**0.209596**	73.182
c_hi_hi	36869.39	**30693.9**	20.119	16123.604	**14756.41**	9.2651	0.49233	**0.47021**	4.7047
i_lo_lo	**25372.92**	29756.08	−14.73	11725.774	**11668.9**	0.48738	0.36828	**0.337766**	9.0350
i_lo_hi	**28745.82**	31660.65	−9.2065	**13523.775**	15943.01	−15.174	**0.43449**	0.441977	−1.6927
i_hi_lo	**21136.15**	30277.17	−30.191	**8495.424**	14451.71	−41.215	**0.29015**	0.448243	−35.269
i_hi_hi	**30867.93**	32186.79	−4.0975	**14913.297**	15059.62	−0.9716	**0.45843**	0.481563	−4.8035
s_lo_lo	23940.33	**22659.19**	5.6539	**8712.591**	10576.13	−17.62	0.30131	**0.210003**	43.48
s_lo_hi	**30302.91**	31204.76	−2.8901	14089.499	**13741.35**	2.53362	**0.44503**	0.47955	−7.1972
s_hi_lo	23675.34	**16966.3**	39.543	8848.901	**6076.776**	45.6184	0.30303	**0.197362**	53.544
s_hi_hi	31235.64	**30463.79**	2.5336	15339.865	**14562.17**	5.34051	0.46957	**0.386948**	21.353
Average			31.058			15.050			23.979

Note. The values in bold indicate that the algorithm provides better results than other algorithm.

Table 3. Comparison of objective values of IDPSO (without VND) with AWS and IDPSO with CWS.

ETC instances	Makespan (s)			Mean flow time (s)			Reliability cost		
	AWS	CWS	RPD	AWS	CWS	RPD	AWS	CWS	RPD
c_lo_lo	13951.21	**13430.5**	3.87662	7020.53	**6795.09**	3.31778	0.20733	**0.20265**	2.3074
c_lo_hi	30016.18	**30005.2**	0.03638	15192.8	**14969.9**	1.48902	0.46494	**0.46113**	0.8252
c_hi_lo	**14058.78**	14398.7	−2.3606	6904.07	**6326.37**	9.13168	0.20665	**0.19164**	7.8327
c_hi_hi	30693.89	**29918.3**	2.59239	14756.4	**14452**	2.10603	0.47021	**0.46353**	1.4408
i_lo_lo	**28666.59**	29087.8	−1.4483	**13296.7**	14253	−6.7094	**0.33465**	0.35936	−6.876
i_lo_hi	**31331.19**	31907.8	−1.8071	**15359.6**	15623.9	−1.6917	**0.4804**	0.49325	−2.606
i_hi_lo	**28214.42**	28275.4	−0.2157	12847.5	**12177.2**	5.50386	**0.40722**	0.41234	−1.2406
i_hi_hi	**31613.98**	31858.3	−0.7669	15440.4	**15423.3**	0.11076	**0.47967**	0.48349	−0.7919
s_lo_lo	26165.1	**25877.5**	1.11112	**10561**	11066.1	−4.5643	0.29675	**0.28117**	5.5396
s_lo_hi	30796.48	**30049.2**	2.48667	15669.9	**15253**	2.73381	**0.47428**	0.47598	−0.3577
s_hi_lo	16966.3	**16397.1**	3.47081	**6076.77**	6144.45	−1.1014	**0.19736**	0.20357	−3.0509
s_hi_hi	32463.79	**32183.1**	0.87187	**14562.1**	14975.2	−2.7581	**0.38694**	0.39186	−1.2554
Average			0.6539			0.6307			0.1473

Note. The values in bold indicate that the algorithm provides better results than other algorithm.

In Table 1, the first column indicates the ETC instance name, the second, third, and fourth columns indicate the makespan, mean flow time, and reliability cost obtained by DPSO with population size 32 (DPSO-32) and DPSO with population size 50 (DPSO-50), respectively. In Tables 1–3, the values in red color indicate an optimal value obtained by the algorithm.

Table 1 shows that the algorithm DPSO-50 gives optimal solutions in majority of the ETC instances compared with DPSO-32. From Table 1, it is inferred that DPSO-50 is able to give better performance in terms of make span by 8.12%, mean flow time by 5.83%, and reliability cost by 4.06% compared with DPSO-32 across all ETC instances, respectively. Figure 3 shows that the DPSO-50 provides minimum fitness value in most of the ETC instances compared with DPSO-32.

Based on Table 1 and Figure 3, the size of the population of the IDPSO algorithm is set to 50. In Table 2, the first column indicates the ETC instance name, the second, third, and fourth columns indicate the makespan, mean flow time, and reliability cost obtained by DPSO-50 and IDPSO, respectively.

The algorithm IDPSO gives optimal solutions in most of the ETC instances compared with DPSO-50,

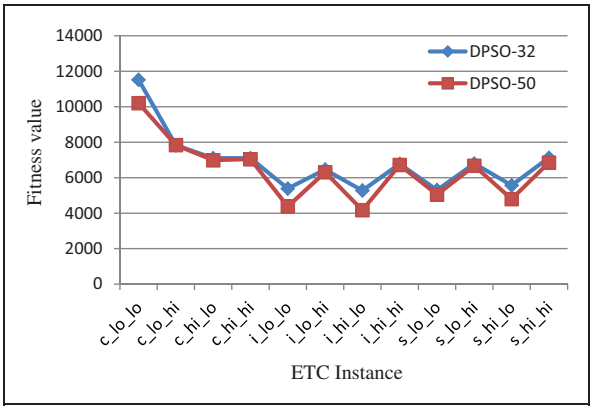

Figure 3. Comparison of fitness value of DPSO-32 and DPSO-50.

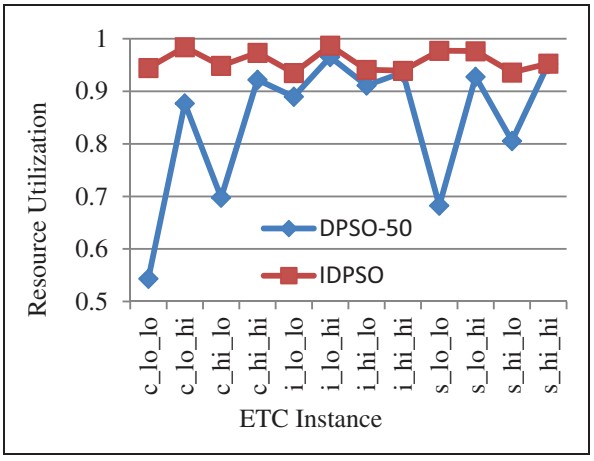

Figure 4. Comparison of RU of IDPSO with DPSO.

which are presented in Table 2. The results obtained from Table 2, the IDPSO is able to provide better performance in terms of make span by 31.06%, mean flow time by 15.05%, and reliability cost by 23.98% compared with DPSO-50 across all ETC instances, respectively. The RU of DPSO-50 and IDPSO are calculated using equation (2) and the values are plotted in Figure 4.

From Figure 4, the RU of IDPSO is in between 0.9 and 1 in most of the ETC instances. The average percentage of RU of IDPSO is 95.7% and DPSO-50 is 84.2% across all ETC instances. The time complexity of the IDPSO is $O(\alpha^2)$. $O(\alpha^2)$ is the time complexity of performing VND heuristic in the flow of IDPSO, where α is the number of tasks assigned to the heavily loaded processor. This heuristic is required not to trap the algorithm in local optima only if the allocation of tasks to the processors by random. In IDPSO, the scheduler performs load balancing using HGL algorithm. The improvement of inclusion of VND heuristic in IDPSO is only by 0.63% in makespan, 2.05% in

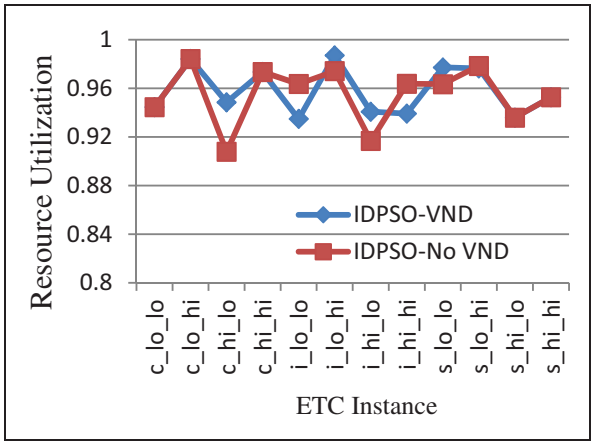

Figure 5. Comparison of RU of IDPSO-VND and IDPSO-No VND.

mean flow time, and 3.09% in reliability cost across all ETC instances. This improvement is negligible compared with the complexity of the VND heuristic. Figure 5 shows the RU comparison of IDPSO with VND and without VND. There is not much more difference in IDPSO with VND and without VND. Therefore, the proposed research work removes the VND in the flow of IDPSO algorithm. After removing of VND in IDPSO, the time complexity of the IDPSO is O(Nnm), where N is the swarm size, n is the number of tasks, and m is the number of processors.

The fitness value of IDPSO is calculated by using AWS method. In this method, the weights of the objective values change generation to generation. The particles in different generation may have the same objective values, but different fitness values. Therefore, the algorithm may take more time to converge and may get stuck in local optima. To address this problem, the proposed work uses constant weights instead of adaptive weights in the fitness function. The Constant Weight Sum (CWS) fitness function is defined in equation (12).

$$\text{Fitness} = 0.4*\text{Makespan} + 0.4*\text{Mean Flow time} + 0.2*\text{Reliability Cost} \quad (12)$$

The performance criteria makespan and flow time have equal importance in independent TS problem. Hence, the weights of the makespan and flow time are set to 0.4 and 0.4, respectively. If the scheduler schedules dependent tasks, then the reliability cost is an important criterion because it will consider both the link reliability and processor reliability. In the proposed work, the scheduler schedules the independent tasks only. Therefore, the reliability cost is less important criterion compared with makespan and flow time

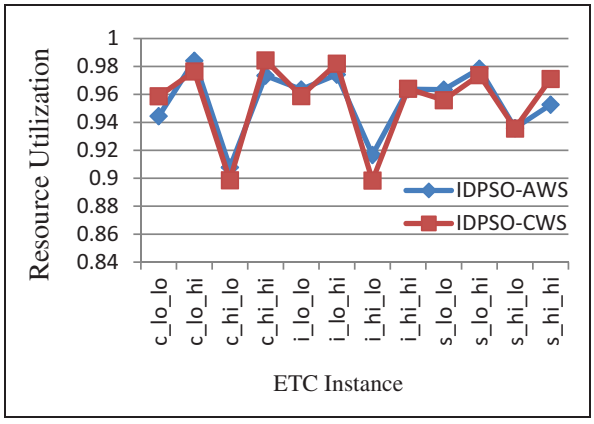

Figure 6. Comparison of RU of IDPSO-AWS and IDPSO-CWS.

Table 4. Average RPD value of IDPSO.

	RPD (%)		
ETC type	Makespan	Mean flow time	Reliability cost
Consistent	96.5195	50.401443	52.32375
Inconsistent	−14.5563	−14.2183	−8.18255
Semi consistent	11.2101	8.968133	27.79495

because it considers only the processor reliability. Hence, the weight of the reliability cost is set to 0.2.

Table 3 shows that the IDPSO-CWS provides optimal solutions in most of the ETC instances for makespan and flow time compared with IDPSO with AWS. The IDPSO-CWS gives less optimal values for reliability cost because of setting low weight age value that is 0.2. The improvement of the IDPSO-CWS compared with IDPSO-AWS of make span increased by 0.65%, mean flow time by 0.63%, and reliability cost by 0.147% across all ETC instances, respectively. Figure 6 shows the comparative performance of RU of IDPSO-AWS and IDPSO-CWS. The average percentage of RU of IDPSO-AWS and IDPSO-CWS is 95.5% across all ETC instances.

The consolidated RPD value of three different ETC types such as consistency, inconsistency, and semi consistency of IDPSO is given in Table 4. The negative value in Table 4 indicates that the algorithm not provides the better result than other algorithms.

From the results obtained in Table 4, the IDPSO algorithm is significantly suitable for consistent ETC instances.

Conclusion

This article presents the problem of scheduling independent tasks to heterogeneous multiprocessor systems using Intelligent DPSO algorithm. The DPSO is a recently developed population-based meta-heuristic technique for discrete optimization problems. The simulation experimental evaluation confirms the efficiency of the incorporation of OPO and HGL techniques into DPSO for scheduling independent tasks in distributed systems to minimize the tri-objectives such as makespan, flow time, and reliability cost. Also, the obtained results presents the IDPSO algorithm is considerably suitable for consistent ETC instances.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

References

1. Mirzayi S and Rafe V. A survey on heuristic task scheduling on distributed systems. In: *2nd world conference on information technology (WCIT-2011)*, Indonesia, AWERProcedia Information Technology & Computer Science, Vol. 1, pp.1498–1501, 2011.
2. Braun TD, Siegel HJ and Bec N. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J Parallel Distrib Comput* 2001; 61: 810–837.
3. Izakian H, Abraham A and Snasel V. Performance comparison of six efficient pure heuristics for scheduling meta-tasks on heterogeneous distributed environments. *Neural Network World* 2009; 19: 695–710.
4. Garshasbi MS and Effatparvar M. Tasks scheduling on parallel heterogeneous multi-processor systems using genetic algorithm. *Int J Comput Appl* 2013; 61: 0975–8887.
5. Kromer P, Snasel V, Platos J, et al. Differential evolution for scheduling independent tasks on heterogeneous distributed environments. *Adv Intell Soft Comput* 2010; 67: 127–134.
6. Zhang L, Chen Y, Sun R, et al. A task scheduling algorithm based on PSO for grid computing. *Int J Comput Intell Res* 2008; 4: 37–43.
7. Kang Q and He H. A novel discrete particle swarm optimization algorithm for meta-task assignment in heterogeneous computing systems. *Microprocess Microsyst* 2011; 35: 10–17.
8. Sarathambekai S and Umamaheswari K. Task scheduling in distributed systems using discrete particle swarm optimization. *Int J Adv Res Comput Sci Software Eng* 2014; 4: 510–522.
9. Kazimipour B, Li X and Qin AK. A review of population initialization techniques for evolutionary algorithms. In: *IEEE congress on evolutionary computation (CEC)*, 2014, pp.2585–2592. Beijing: IEEE.

10. Tizhoosh HR. Opposition-based learning: A new scheme for machine intelligence. *Int Conf Comput Intell Model Control Autom* 2005; 1: 695–701.
11. Tizhoosh HR. Opposition-based reinforcement learning. *J Adv Comput Intell Intell Inform* 2006; 10: 578–585.
12. Rahnamayan S, Tizhoosh HR and Salama MMA. Opposition-based differential evolution. *IEEE Trans Evol Comput* 2008; 12: 64–79.
13. Iqbal A, Jabeen H and Baig R. Opposition based genetic algorithm with jumping phenomena. In: *2nd international symposium on intelligent informatics*, Canada, 2009, pp.113–120.
14. Wang H and Liu Y. Opposition-based particle swarm algorithm with Cauchy mutation. In: *IEEE congress on evolutionary computation*, 2007, pp.4750–4756.
15. Omran MGH. Using opposition-based learning with particle swarm optimization and barebones differential evolution. *Comput Inform Sci* 2009; 1: 373–384.
16. Ergezer M. Oppositional biogeography-based optimization for combinatorial problems. In: *IEEE congress on evolutionary computation*, New Orleans, LA, 2011, pp.1496–1503.
17. Alharbi F. Simple scheduling algorithm with load balancing for grid computing. *Asian Trans Comput* 2012; 2: 8–15.
18. Lindeke R. (2005) Scheduling of Jobs, IE 3265 -POM, Spring [online] http://www.docfoc.com/scheduling-of-jobs (accessed 28 June 2016).
19. Kim IY and de Weck OL. Adaptive weighted sum method for multi-objective optimization: a new method for Pareto front generation. *Struct Multidiscip Optim* 2006; 31: 105–116.
20. Ali S and Siegel HJ. Representing task and machine heterogeneities for heterogeneous computing systems. *Tamkang J Sci Eng* 2000; 3: 195–207.
21. Qin X and Jiang H. Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems. In: *IEEE international conference on parallel processing*, Valencia, Spain, 2001, pp.113–122.