Population-based Metaheuristics for Tasks Scheduling in Heterogeneous Distributed Systems

Flavia Zamfirache, Marc Frîncu, Daniela Zaharie

Department of Computer Science, West University of Timişoara, Romania
{zflavia,mfrincu,dzaharie}@info.uvt.ro

Abstract. This paper proposes a simple population based heuristic for task scheduling in heterogeneous distributed systems. The heuristic is based on a hybrid perturbation operator which combines greedy and random strategies in order to ensure local improvement of the schedules. The behaviour of the scheduling algorithm is tested for batch and online scheduling problems and is compared with other scheduling heuristics.

1 Introduction

Since the work of Braun et al. [1] which illustrated the fact that genetic algorithms can generate good solutions for task scheduling problems, a lot of other population-based metaheuristics were proposed (e.g. evolutionary algorithms [2], ant systems [7], memetic algorithms [10]). Unlike the genetic algorithm in [1] which is based on classical mutation and crossover operators, the recent approaches use specific local search operators. Most researchers identified as effective operators those involving a rebalancing of the load on different processors by moving or swapping tasks between processors. Currently there exist both simple and sophisticated "rebalancing" operators. The aim of this paper is to identify the basic components of such operators and to design a simple population-based scheduler involving as few as possible search mechanisms.

The addressed problem is that of assigning a set of independent and non-preemptive tasks to a set of resources (e.g. machines, processors) such that the maximal execution time over all resources, i.e. *makespan*, is minimized. The assignment of tasks is based on estimations of the execution times of the tasks on various resources. Let us consider a set of n tasks, $\{t_1, \ldots, t_n\}$, to be scheduled on a set of $m < n$ processors, $\{p_1, \ldots, p_m\}$. Let us suppose that for each pair (t_i, p_j) we know an estimation $ET(i, j)$ of the time needed to execute the task t_i on the processor p_j. A schedule is an assignment of tasks to resources, $S = (p_{j_1}, \ldots, p_{j_n})$, where $j_i \in \{1, \ldots, m\}$ and p_{j_i} denotes the processor to which the task t_i is assigned. If $\mathcal{T}_j$ denotes the set of tasks assigned to processor p_j and T_j^0 denotes the time moment since the processor j is free then the completion time corresponding to this processor will be $CT_j = T_j^0 + \sum_{i \in \mathcal{T}_j} ET(i, j)$. The makespan is just the maximal completion time over all processors, i.e.

$makespan = \max_{j=\overline{1,m}} CT_j$. The problem to be solved is that of finding the schedule with the minimal makespan value. In real distributed systems tasks arrive continuously and have to be assigned to resources either as they arrive or when a scheduling event is triggered. In this work we analyze both the case when the scheduling event is triggered when a given number of tasks have arrived (batch scheduling) and the case when the scheduling is activated at pre-specified moments of time (online scheduling). The main idea of the proposed population-based heuristic is described in Section 2. Sections 3 and 4 present the numerical results obtained for batch and online scheduling problems while Section 5 concludes the paper.

Table 1. Characteristics of the strategies used to construct initial schedules (Notations: CT - current completion time, ET - execution time, ECT - estimated completion time)

Task selection	Processor selection	Strategy
Random	Random	Random
Random	Min CT	Opportunistic Load Balancing (OLB)
Random	Min ET	Minimum Execution Time (MET)
Random	Min ECT	Minimum Completion Time (MCT)
Increasing min ECT	Min ECT	MinMin
Decreasing min ECT	Min ECT	MaxMin

2 Designing Heuristics for Task Scheduling

The construction of a (sub)-optimal schedule is usually based on creating an initial schedule which is then iteratively improved. When constructing an initial schedule there are two decisions to take: (i) the order in which the tasks are assigned to processors; (ii) the criterion used to select the processor corresponding to each task. Depending on these elements there exist several strategies [1] as presented in Table 1. Each of these strategies generates initial schedules with a specific potential of being improved. Therefore it would be beneficial to use not just one strategy but to use a population of initial schedules constructed through different strategies.

The initial schedules created by the scheduling heuristics are usually non-optimal and thus they can be improved by moving or swapping tasks between resources. Depending on the criteria used to select the source and destination resources, and the tasks to be relocated there can be designed a lot of strategies to perturb a schedule [9]. Most perturbation operators involved in the scheduling heuristics used in task scheduling are based on two typical operations: "move" one task from a resource to another one and "swap" two tasks between their resources. In order to obtain an immediate improvement in the schedule, the most loaded resource (which determine the makespan) should be involved in the operation. The largest improvement can be obtained by an exhaustive search for

Table 2. Characteristics of the strategies used to perturb the schedules

Source		Destination		Strategy
Processor	Task	Processor	Task	
Random	Random	Random	-	*Random Move*
Most loaded (max CT)	Random	Best improvement	-	*Greedy Move*
Most loaded (max CT)	Random	Least Loaded (min CT)	Random	*Greedy Swap*

the pair consisting of the task to be moved and the destination processor. Besides the fact that this operation is costly ($\mathcal{O}(mn)$) it can fail to generate in just one step a schedule with a smaller makespan. For instance, if there are several processors reaching the maximal completion time it is necessary to apply for several times the "move" operation in order to obtain a decrease of the overall makespan. On the other hand if there is no pair (task, destination processor) which allows to decrease the makespan of the source processor then the "swap" operation should be used instead. In the case of an exhaustive search for the pair of tasks to be swapped the complexity order could be in the worst case $\mathcal{O}(n^2)$ which for a large number of tasks becomes impractical. Therefore from the large number of possible choices of source and destination processors and of tasks to be relocated we selected those which do not involve a systematic search in the set of tasks (i.e. the tasks to be relocated are randomly chosen). The strategies presented in Table 2 were selected based on their simplicity, efficiency and randomness/greediness balance. The "random move" corresponds to the "local move" operator [10] and is similar to the mutation operator used in evolutionary algorithms. The "greedy move" operator is related to the "steepest local move" in [10] but with a higher greediness since it always involves the most loaded processor. The "greedy swap" is similar to "steepest local swap" in [10] but it is less greedy and less expensive since it does not involve a search over the set of tasks.

Since one perturbation step does not necessarily lead to an improvement in the quality of a schedule we consider an iterated application of the perturbation step until either n iterations were executed (each task has the chance to be moved) or a maximal number, g_p, of unsuccessful perturbations is reached.

The influence of g_p on the quality of the schedule is analyzed in the next section. On the other hand in order to exploit the search abilities of each strategy it seems natural to combine several perturbation operators. Thus the strategies in Table 2 are combined as described in the Algorithm 1 (*HybridPerturbation*). This hybrid perturbation has a structure similar to the "re-balancing" mutation described in [10]. However there are some differences between them. In [10] the "swap" perturbation is applied before "move" perturbation while in the hybrid perturbation described in Algorithm 1 the order is reversed. This apparently minor difference influences the overall cost of the perturbation as the application of the "move" operation is less costly than that of "swap" and it can induce a larger gain in the makespan. On the other hand in [10] only one perturbation step is applied to a schedule at each evolutionary generation. Moreover in the

Algorithm 1 The general structure of the population based scheduler

SimplePopulationScheduler (SPS)

Generate the set of initial schedules:
$S \leftarrow \{S_1, \ldots, S_N\}$
while ⟨the stopping condition is false⟩ **do**
  for $i = \overline{1, N}$ **do**
    $S'_i \leftarrow$perturb(S_i)
  end for
  $S \leftarrow$ select$(S, \{S'_1, \ldots, S'_N\})$
end while

SimplePerturbation(S)

$i \leftarrow 0$; fail$\leftarrow 0$
while $i < n$ and fail$< g_p$ **do**
  $i \leftarrow i + 1$
  if GreedyMove/Swap(S) is successfull **then**
    fail$\leftarrow 0$; S$\leftarrow$GreedyMove/Swap(S)
  else
    fail$\leftarrow$fail+1
    if random$(0, 1) < p_m$ **then**
      S$\leftarrow$ RandomMove(S)
    end if
  end if
end while
return S

HybridPerturbation(S)

$i \leftarrow 0$; fail$\leftarrow 0$
while $i < n$ and fail$< g_p$ **do**
  $i \leftarrow i + 1$
  if GreedyMove(S) is successfull **then**
    fail$\leftarrow 0$; S$\leftarrow$GreedyMove(S)
  else
    if GreedySwap(S) is successfull **then**
      fail$\leftarrow 0$; S$\leftarrow$GreedySwap(S)
    else
      fail$\leftarrow$fail+1
      if random$(0, 1) < p_m$ **then**
        S$\leftarrow$ RandomMove(S)
      end if
    end if
  end if
end while
return S

"re-balancing" operator the random perturbation is applied any time when the "swap"-"move" duo is unsuccessful while in our case the random perturbation is interpreted as a mutation, thus it is applied with a small probability (e.g. $p_m = 1/n$).

Having the perturbation as key operator we designed a simple population-based heuristics described in Algorithm 1 (*SPS* - *SimplePopulationScheduler*). Besides the perturbation operator which can be a simple (*SimplePerturbation*) or a hybrid one (*HybridPerturbation*) there are two other elements which can influence the behaviour of the algorithm: initialization and selection. The use of some seed schedules in the initial population has been emphasized by many authors [1, 6, 10]. Consequently, besides the plain random schedules we included in the initial population also schedules generated with the heuristics listed in Table 1. During the iterative process, each schedule, S_i, in the current population is perturbed leading to a new schedule S'_i (it should be mentioned that in the case of unsuccessful perturbation, S_i could remain unchanged). The schedules corresponding to the next iterative step (generation) are selected from the sets of current and perturbed schedules using a binary tournament approach (the schedule with the smallest makespan from a randomly selected pair of schedules is selected). To ensure the elitism, the best element of the population is preserved.

A preliminary analysis on the role of crossover in generating good schedules illustrated that no significant gain is obtained by using crossover (at least uniform and one cut-point crossover). Since the number of processors is usually significantly smaller than the number of tasks almost all processors are involved in the schedules included in the population. Thus the set of schedules generated by a crossover operator would not be significantly different from the set of schedules which could be generated by applying only the iterated perturbation.

3 Numerical Results for Batch Scheduling

Let us consider the case where the scheduling event is activated when a given number of tasks arrived to the scheduler. This is a classical batch scheduling problem characterized by the fact that some data concerning the estimated execution time of tasks on different resources is known. As test data we have used those introduced in [1] which provides matrices containing values of the expected computation time (ET) generated based on different assumptions related to tasks and resources heterogeneity (low and high) and consistency (consistent, semi-consistent and inconsistent). The data correspond to the case of 512 tasks to be scheduled on 16 processors.

The aim of the numerical study was to analyse the influence of the perturbation strategies on the performance of a Simple Population-based Scheduler (SPS) having the structure described in Algorithm 1. The parameters involved in the algorithm were set based on preliminary parameters tuning leading to the following values: (i) 25 elements in the population (populations of sizes 50, 100 and 200 were also analysed); (ii) a maximal number of successive failures (g_p) in the perturbation operator equal to 150 (values between 10 and 300 were tested; the influence of this parameter on the performance of the scheduler is illustrated in Figure 1 for three test cases); (iii) a probability of applying random perturbations (p_m) equal to $1/n \simeq 0.002$. The maximal number of iterations involved in the stopping conditions was set to 8000. This is in accordance with the values used in literature for evolutionary schedulers [1]. The average time needed to generate a schedule is around 40s (on a Intel P8400 at 2.26GHz) which is also consistent with the time reported in [10] (90s on a AMD K6(tm) at 450MHz).

The analysed initialization strategies are: (i) random initialization; (ii) use of the scheduling heuristics described in Table 1 and randomly initialize the other elements; (iii) use random perturbations of the scheduling heuristics in Table 1; (iv) use the MinMin heuristic and random perturbations of this. As expected, the best results were obtained when the initial population contains seeds obtained by using scheduling heuristics while the worst behaviour corresponds to purely random initialization. The numerical results presented in Table 3 correspond to the three perturbation variants (move-based, swap-based and the hybrid one) and to a state of the art memetic algorithm hybridized with Tabu Search (MA+TS) [10]. Even if based on simpler operators, the algorithm proposed in this work provides schedules close in quality to those generated by MA+TS. Moreover in

Table 3. Averages and standard deviations (computed by 30 independent runs) of the makespan obtained by the population-based scheduler with different perturbation strategies. The best and the second best values (validated by a t-test with a significance level of 0.05) for each problem are in bold and in italic, respectively.

Pb.	GreedyMove	GreedySwap	Hybrid	MA+TS[10]
u_c_hihi	7684852.40(±24798)	7689131.76(±26971)	*7609663.13*(±30673)	**7530020.18**
u_c_hilo	155248.33(±551)	155495.10(±158)	*154979.43*(±180)	**153917.17**
u_c_lohi	251445.60(± 809)	250558.63(±1028)	*248903.70*(±1014)	**245288.94**
u_c_lolo	5255.06(±8.90)	5258.4(±7.65)	*5235.00*(±5.32)	**5173.72**
u_i_hihi	3072453.70(±18667)	**3019756**(±14323)	**3014083.63**(±21420)	*3058474.90*
u_i_hilo	75222.90(±318.46)	*74684.433*(±225.12)	**74553.20**(±130.78)	75108.49
u_i_lohi	106309.56(±706.70)	*105261.20*(±561.41)	**105013.60**(±516.63)	105808.58
u_i_lolo	2617.26(±16.04)	*2590.83*(±8.18)	**2585.70**(±6.05)	2596.57
u_s_hihi	4382845.80(±50248)	4352017.96(±36899)	**4316556.23**(±29236)	**4321015.44**
u_s_hilo	*98036.16*(±241.10)	98302.366(±363.54)	*97964.86*(±364.56)	**97177.29**
u_s_lohi	127565.00(±613.97)	*127026.63*(±499.45)	**126763.23**(±564.75)	127633.02
u_s_lolo	3538.96(±19.28)	3526.53(±11.24)	*3520.80*(±11.39)	**3484.08**

the case of inconsistent test cases ("u_i_**" problems) the proposed scheduler using the hybrid perturbation operator provides better results.

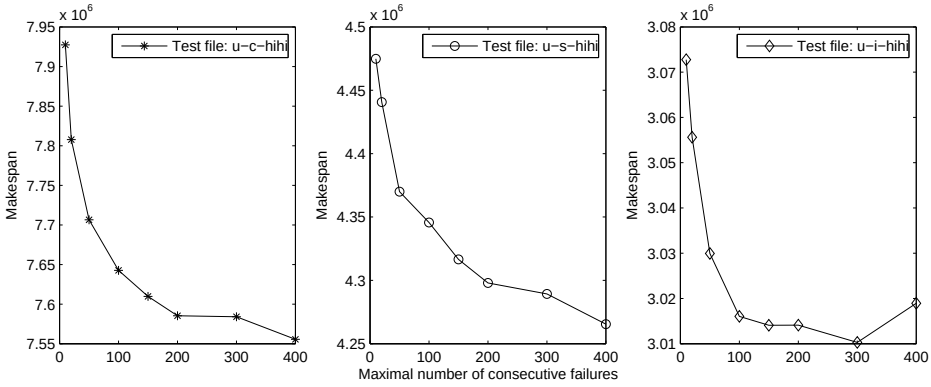

Fig. 1. Influence of the maximal number of consecutive mutations without improvement (g_p) on the makespan.

4 Numerical Results for Online Scheduling

For online scheduling we considered a simulation model where task executions times (ET) follow a Pareto distribution with $\alpha = 2$ and the tasks arrival rate is modelled based on statistical results extrapolated from real world traces [3].

A total number of 500 tasks were generated for every test. Rescheduling was done every 250 time units given a minimal execution time of 1000 units. All tests were repeated 20 times in order to collect statistics. The main aim of the numerical tests was to analyze if using populations of schedules one can obtain improvements in the quality with an acceptable loss in the scheduling time. Therefore several dynamic scheduling heuristics with ageing have been tested against their corresponding population based versions which were constructed by using the specific scheduling heuristics as perturbation operators in SPS. Their behaviour has also been compared with the SPS algorithm based on a non-iterated hybrid perturbation (at each perturbation step the hybrid perturbation is applied only once). Among the online scheduling algorithms we tested a flavour of DMECT as described in [4], the MinQL heuristic [5] and the classic MinMin and MaxMin with ageing. DMECT periodically computes for every task the Local Waiting Time (LWT) - the time since it was assigned to the current processor queue - and a σ value that depends on the implementation and could take into account the estimated execution time (ET). This paper uses the values given in [4]. From these values a decision on whether to move the task or not is taken by checking if the $\sigma - LWT$ is smaller than 0 or not. MinQL allows for optimal balancing the tasks inside resources while taking into account both the age of the task and the priority of local tasks. It uses a backfilling approach where multiple selection conditions for the destination resource can be used. The version used for testing in this paper uses a selection based on the CPU speed.

The population variants of the two previously mentioned scheduling heuristics use a population of 25 elements initialized both with random schedules (60%) and by using the MinMin heuristics (40%). The scheduling heuristics is then applied on every element to generate perturbed schedules and the surviving elements are selected by tournament. The procedure stops when an improvement in the makespan of at least 10% is no longer noticed after a given number of iterations (e.g. 600). Table 4 presents the main benefits of population based scheduling heuristics (pDMECT and pMinQL) when used in online scheduling. Both pDMECT and pMinQL obtained significantly better results than their non-populational variants, with pDMECT having a behaviour similar to SPS (the best values in Table 4 are bold-faced and they were validated using a t-test with 0.05 as level of significance). The only notable difference in the behaviour of pDMECT and SPS was that of speed. pDMECT required almost 30 seconds to build a schedule while the simple population-based scheduler needed only three seconds on average. The reason for this difference lies in the complexity of one scheduling step in pDMECT, $\mathcal{O}(m \times n)$, compared with that of one perturbation step in SPS, $\mathcal{O}(m)$, where m represents the number of processors and is significantly smaller than n which is the number of tasks.

5 Conclusions

The simple population-based scheduler using an iterated hybrid perturbation operator provides solutions to batch scheduling problems which are compara-

Table 4. Average makespan (MS) obtained by online scheduling heuristics and their population based variants

	DMECT	pDMECT	MinQL	pMinQL	SPS	MaxMin	MinMin
MS	66556.20± 15097.85	**49409.11**± 9522.13	76564.40± 18114.51	54332.89± 9891.15	**46996.76**± 8812.87	61165.15± 11936.19	68774.87± 15101.05
Time (ms)	66.56 ± 15.50	28343.04± 10702.15	3.06 ± 2.52	2254.64± 314.45	2777.70± 578.22	684.49± 242.15	669.21± 209.99

ble in quality with those generated by schedulers using more sophisticated local search operators [10]. The main benefit is obtained in the case of highly heterogeneous and inconsistent distributed environments. The idea of using a simple population-based heuristic proved to ensure a good compromise between solution quality and computational cost also in the case of online scheduling. Further work will address the case of interrelated tasks and that of using other metrics such as the *Total Processing Consumption Cycle* which is an alternative to the makespan and is independent of the hardware.

Acknowledgments. This work is supported by Romanian project PNCD II 11-028/ 14.09.2007 (NatComp).

References

1. T.D. Braun, H.J. Siegel, N. Beck et al., A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. Journal of Parallel and Distributed Computing, 61(6), pp. 810-837, 2001.
2. J. Carretero, F. Xhafa, Using Genetic Algorithms for Scheduling Jobs in Large Scale Grid Applications. Journal of Technological and Economic Development - A Research Journal of Vilnius Gediminas Technical University, 12(1), pp. 11-17, 2006.
3. D. G. Feitelson, Workload modeling for computer systems performance evaluation, http://www.cs.huji.ac.il/ feit/wlmod/, 2010.
4. M. Frincu, Dynamic Scheduling Algorithm for Heterogeneous Environments with Regular Task Input from Multiple Requests. In LNCS 5529, pp. 199–210, 2009.
5. M. Frincu, G. Macariu, A. Carstea, Dynamic and Adaptive Workflow Execution Platform for Symbolic Computations. Pollack Periodica, Akademiai Kiado, 4(1), pp. 145–156, 2009.
6. A.J. Page, T.M. Keane, T.J. Naughton, Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneus distributed system. J. Parallel Distrib. Comput., doi:10.1016/j.jpdc.2010.03.11, 2010.
7. G. Ritchie, J. Levine, A hybrid ant algorithm for scheduling independent jobs in heterogeneous computing environments. In Proc. of 23rd Workshop of the UK Planning and Scheduling Special Interest Group, 2004.
8. A.J Page,T. J. Naughton, Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In Proc. of 19th IEEE/ACM International Parallel and Distributed Processing Symposium, Denver, pp. 1530-2075, 2005.
9. F. Xhafa, A. Abraham, Computational models and heuristic methods for Grid scheduling problems. Future Generation Computer Systems 26, pp. 608-621, 2010.
10. F.Xhafa, A Hybrid Evolutionary Heuristic for Job Scheduling on Computational Grids, in Hybrid Evolutionary Algorithms, LNCS 75, pp.269-311, 2007.