# Resource Management in Distributed and Parallel Systems: Paper Review of Parallel Machine Scheduling with Dynamic Resource Allocation via a Master – Slave Genetic Algorithm

alexandru.ionascu96@e-uvt.ro

January 2020

## 1 Introduction

This paper[1] addresses the problem of parallel machine scheduling in a dynamic resource allocation context. The objective function is minimizing the makespan where jobs can benefit from additional resources thus decreasing the processing time. To minimize that, we need to establish job sequences, assignment, their starting time, and also the allocation of resources.

A master-slave genetic algorithm is presented to solve three of this problem, and an intuitive greedy heuristic is treated as a separate task to determine the job starting time. The paper includes several benchmarks comparing this method to other well-known algorithms and proves to be a strong competitor. The genetic algorithm is presented generically, and many more improvements are expected to arise in practical situations.

The results are reliable since the variables don't present a strong dependency and we expect it to adopt this method in most of the situations we may encounter. In this paper review, we will cover the problem's encoding, full details on the approached method and as well as a brief Python 3 implementation.

## 2 Master-Slave Genetic Algorithm

The main difference between the presented approach and a standard genetic algorithm is that instead of a singular chromosome representation we will have a separate master representation for resource allocation problem and a separate slave population to represent the job assignments and their sequence.

The paper includes further description of the mutation and slave-crossover phases, and the latter evolution steps are treated rather in a generic way. The fitness evaluation is treated separated and a heuristic is provided for job starting times.

## 3 Population Initialization

At the master level, we simply generate random processing modes to represent resource allocation. However, the slaves for each master are generated based on yet another heuristic to speed up the process. We consider the resource allocation initially fixed and thus we generate a first candidate for a slave encoding on the longest-processing time job first criteria. The rest of the slaves are based on the initial slave with lambda interchange mutations.

```
def init_master():
  return [randint(1, modes[job - 1]) for job in jobs]

def init_slave(master):
  # sort by longest proc_time first
  p_times = [-1 * proc_time[i][master[i] - 1]
    for i in range(len(master))]
  arg_sort_jobs = np.argsort(p_times)
  chr = []
  for i in range(len(arg_sort_jobs)):
    chr.append(str(jobs[arg_sort_jobs[i]]))
    if i % (M + 2) == 0:
      chr.append('*')
  return chr

def lambda_interchange(slave):
  i, j  = sample(range(len(slave)), 2)
  i, j = (j, i) if i > j else (i, j)

  # try again
  if slave[i] == '*' or slave[j] == '*'  or j == len(slave) - 1:
    return lambda_interchange(slave)
  i_end = i + 1
  j_end = j + 1

  for k in range(i, len(slave) - 1):
    if slave[k + 1] == '*' or k == j-1:
      break
    i_end += 1

  for k in range(j, len(slave) - 1):
    if slave[k] == '*':
      break
    j_end += 1

  return slave[:i] +
```

```
        slave[j:j_end] +
        slave[i_end : j] +
        slave[i:i_end] + slave[j_end:]

NP = 10 # population size
NS = 10 # number of slaves per master
def init_population():
  master = init_master()
  slave = init_slave(master)
  population = [lambda_interchange(slave) for _ in range(NS)]
  return (slave, population)

population = [init_population() for _ in range(NP)]
```

# 4 Master Mutation

It's difficult to perform any other operation such as crossover on resource allocation, thus we only have basic mutations. We choose two random processing modes and swap them with the existing ones.

```
MUTATION_RATE = 0.75
def mutate_master(master):
  # select random job and assign random mutation
  if random() >= MUTATION_RATE:
    return master
  mutate_idx = randint(0, len(master) - 1)
  master[mutate_idx] = randint(1, modes[mutate_idx])
  return master
```

# 5 Slave Mutation

A similar simplistic operation can be applied to the slave encoding. It's the so-called roulette style of choice between two candidates and we swap their positions in the job sequence list. The list also contains special delimiters to represent the end of the assignment for the current machine, and thus, the swapping produces also different job assignments besides different sequencing.

```
def mutate_slave(slave):
  # perform swap mutation
  if random() >= MUTATION_RATE:
    return master
  i, j  = sample(range(len(slave)), 2)
  slave[i], slave[j] = slave[j], slave[i]
```

```
  for i in range(1, len(slave)):
    if slave[0] == '*' or
       slave[-1] == '*' or
       (slave[i - 1] == slave[i] and slave[i] == '*'):
        mutate_slave()
        break
```

## 6 Slave Crossover

It is an already well-known method named partial matching crossover. We choose a selection between two random points and we form a partial permutation that is applied in along both slaves. The selected regions are initially swapped.

```
def cross_over_slaves(slave_1, slave_2):
  # apply partial cross over mapping
  i, j  = sample(range(len(slave_1)), 2)
  i, j = (j, i) if i > j else (i, j)

  for k in range(i, j + 1):
    if slave_1[k] == '*' or slave_2[k] == '*':
      # try again
      cross_over_slaves(slave_1, slave_2)
      return

  p_map = {}

  for k in range(i, j + 1):
    p_map[slave_1[k]] = slave_2[k]
    p_map[slave_2[k]] = slave_1[k]

  for k in range(len(slave_1)):
    if slave_1[k] in p_map:
      slave_1[k] = p_map[slave_1[k]]

    if slave_2[k] in p_map:
      slave_2[k] = p_map[slave_2[k]]
```

## 7 Fitness

We described briefly the main evolutionary operators. Now we describe the fitness function and the heuristic for the starting times. We first calculate the machines available at a certain time. If we have available resources we simply

assign the corresponding job index on the machines for further processing. Otherwise, we calculate the left workload for each machine and we define a priority criterion based on the sum of left workload and the ration between processing time and resource allocation for the current job for each machine.

```
def calculate_fitness(master, slave):
  NaN = float('nan')
  t = 0
  # start indexing by 1
  # p[i] - processing time for job i
  p = [NaN] + [proc_time[i][master[i] - 1] for i in range(len(master))]
  # r[i] - resources taken for job i

  res = [NaN] + [r[i][master[i] - 1] for i in range(len(master))]

  # set of scheduled jobs on machine i
  scheduled = [[NaN]] + [[NaN] for i in range(M)]
  # number of unscheduled jobs on machine i
  unscheduled = [[NaN]] + [[NaN] for i in range(M)]

  # index o of the current assign job on machine j
  sigma = [NaN] + [1 for j in range(M)]

  # pi[j] = i -- job i assigned on machine j
  pi = [[NaN]] + [[NaN] for i in range(M)]
  machine_idx = 1
  for job in slave:
    if job == '*':
      machine_idx += 1
    else:
      unscheduled[machine_idx].append(int(job))
      pi[machine_idx].append(int(job))

  # the available time on machine i
  c = [NaN] + [0 for i in range(M)]
  # starting times
  tau = [NaN] + [NaN for i in range(N)]

  Rt = R - sum([res[scheduled[j]] \
                for j in range(1, M + 1) \
                for i in range(1, len(scheduled[j]))])

  while True:
    # obtain the available machines at time t
    Mt = [NaN]
```

```
  for j in range(1, M + 1):
    if len(unscheduled[j]) == 1: # default is one Nan elem
      continue
    if res[pi[j][sigma[j]]] <= Rt and c[j] <= t:
      Mt.append(j)


  while len(Mt) > 1:
    if sum([res[pi[j][sigma[j]]] for j in Mt if j == j]) <= Rt:
      # every available machine is aranged
      for j in range(1, len(Mt)):
        tau[pi[j][sigma[j]]] = t
        c[j] = t + p[pi[j][sigma[j]]]
        scheduled[j].append(pi[j][sigma[j]])
        unscheduled[j].remove(pi[j][sigma[j]])
        sigma[j] += 1
    else:
        # calculate left workload
        P = [NaN] + [sum([i for i in unscheduled[j] if i == i])
          for j in Mt if j == j] # ignore NaN

        # calculate priority
        w = [0] + [P[j] + p[pi[j][sigma[j]]] / res[pi[j][sigma[j]]]
          for j in Mt if j == j]

        j_prime = np.argmax(w)
        j_prime = Mt[j_prime]
        tau[pi[j_prime][sigma[j_prime]]] = t
        c[j_prime] = t + p[pi[j_prime][sigma[j_prime]]]
        scheduled[j_prime].append(pi[j_prime][sigma[j_prime]])
        unscheduled[j_prime].remove(pi[j_prime][sigma[j_prime]])
        Rt -= res[pi[j_prime][sigma[j_prime]]]
        to_remove = [j_prime] + [Mt[j] for j in range(1, len(Mt))
          if res[pi[j_prime][sigma[j_prime]]] > Rt]

        for elem in to_remove:
          Mt.remove(elem)
        sigma[j_prime] += 1

if scheduled == pi:
  return max([c[j] for j in range(1, M + 1)])
```

## 8 Evolutionary process

As a stopping criterion, we choose a max-iteration approach. The method is non-elitist since we don't discard actual solution but we use in-place operations among the solution encoding. The whole evolutionary scheme cand be mostly adapted to any other problems since the authors presented only a very high-level sketch of this step.

```
def run_algorithm():
  MAX_ITER = 100
  Best = None
  population = init_population()
  CRs = len(population) / 2
  for _ in range(MAX_ITER):
    fitness = [
        calculate_fitness(master, slave)
        for (master, slave) in population
    ]
    for (master, slave) in population:
        mutate_master(master)
        mutate_slave(slave)
    for _ in range(CRs):
        (m1, s1), (m2, s2) = sample(population, 2)
        cross_over_slaves(s1, s2)
    Best = max(fitnesses)
```

## 9 Results

The proposed algorithm was tested against Differential Heuristic and Tabu Search. The experimental results showed big improvements compared to those methods. Other tests were made between the master-slave genetic algorithm and the standard genetic algorithm. In most cases, the master-slave genetic algorithm outperformed the standard one, however, we can't say that the difference between them was large.

## 10 Conclusion

We analyzed an interesting idea of the master-slave genetic algorithm. The separation between tasks was very clear and made the solution relatively simple. We followed a possible implementation in Python 3. More optimizations are yet possible but in the current state, it outperforms some of the comparative heuristics.

The evolutionary process can be largely adapted to many more problems, as in terms of variable initializations, the paper presents sound results by fixing

only two-three variables. In the subfield of genetic algorithms, we can come up with many tricks regarding problem-specific knowledge. In this way, the presented article can become a solid base for a top-performing algorithm in practical situations.

# References

[1] Yaping Fu, Guangdong Tian, Zhiwu Li, and Zhenling Wang. Parallel machine scheduling with dynamic resource allocation via a master-slave genetic algorithm: Parallel machine scheduling with dynamic resource allocation. *IEEJ Transactions on Electrical and Electronic Engineering*, 13, 01 2018.