3

Proposed Algorithms for Grid Resource Scheduling

3.1 Introduction

Computational Grid is a distributed and heterogeneous computing system with a number of resources of different types. Since a worldwide grid could span over million of computers connected via Internet so grid resource scheduling becomes complex as the size of the grid grows. Mapping a set of tasks on to set of resources in this environment is compute intensive problem. Dynamic nature of resources, different administrative policies and Quality of Service (QoS) requirements make the

problem even tougher. Use of conventional methods becomes infeasible as the search space becomes very large. A new area of research has been setup to tackle this problem, which makes use of meta-heuristic techniques to find the optimal or near optimal solution to the problem. This chapter proposes and discusses in detail, three new algorithms for resource scheduling on computational grids. These are hybrid algorithms based on well-known meta-heuristics Genetic Algorithms, Ants' Colony Optimization and Simulated Annealing.

Many algorithms exist [80], [81], [84]-[89], [93]-[95], [100]-[103] in literature based on these techniques for resource scheduling. The proposed techniques are different from existing ones in many ways. First, most of the existing algorithms are not particularly designed and tested for grid systems. Secondly, the algorithms that are tested in grid environment consider only makespan as Quality of Service (QoS) parameter. They fail to take the user interests into consideration, which is one of the important QoS parameter of grids. So, proposed algorithms differ in the objective / fitness function from the existing ones. Proposed algorithms try to find the optimal schedule based on the objective that ensures the proper utilization of the resources as well as try to respects the interests of grid users by minimizing the delay in meeting deadlines for the grid jobs.

3.2 Problem Formulation

Grid scheduler is the important part of Grid Resource Management System (GRMS), which gathers the information about the resources and chooses the best resource as per the job requirements. This is followed by the actual execution of the jobs. The problem of finding the best "*job-resource*" pair is a compute-intensive problem and need to be formulated mathematically to find the optimal solution.

To formulate the problem, we consider mapping a set of independent user jobs $\{J_1, J_2, J_3, \ldots, J_N\}$ to a set of heterogeneous processors / resources $\{P_1,P_2,P_3,\ldots,P_M\}$ (Though we can have any type of resource that can be shared on a computational grids, but for our simulations we have considered only processors as a resource.) This mapping is done with an objective of minimizing the completion time and utilizing the resources effectively, and also minimizing the delay in meeting user specified deadlines. The speed of each resource is expressed in number of cycles per unit time, and the length of each job in number of cycles. Each job J has processing requirement C_j cycles and processor P_i has speed of V_i cycles/second. Any job J has to be processed by P_i, until completion.

3.2.1 Objective / Fitness Function

The Objective / fitness function is used to differentiate between high and low quality solutions. It is defined in terms of objectives of the problem in hand. For grid scheduling problem, the users have goal of satisfying the deadlines of jobs submitted by them whereas the resource providers like to minimize the makespan. The schedule S is evaluated on the basis of fitness function, which is defined as

$$\textbf{Fitness, } F(S) = 1/(\omega * MS + \theta * T_{delay}) \quad (3.1)$$

Where MS denotes the makespan of the schedule and T_{delay} denotes the cumulative delay in meeting deadlines. ω $(0 \leq \omega \leq 1)$ and θ $(0 \leq \theta \leq 1)$ are the weights to prioritize the components of the fitness function as per the needs of the resource provider and job users. The components MS and T_{delay} of the fitness function are defined as follows.

Let $t_{end}(i)$ and $t_{dline}(i)$ be turnaround time and deadline time for i_{th} job, then

$$t_{delay}(i) = t_{end}(i) - t_{dline}(i) \quad (3.2)$$

{t_{delay} is delay in meeting deadline for i_{th} job, and

$t_{delay} = 0$ *if* $t_{end} < t_{dline}$ } and

$$T_{delay} = \Sigma t_{delay}(i) \qquad for\ 1<i<N \tag{3.3}$$

Makespan of schedule *S* is defined as

$$MS = Max\ (t_{processing}(k)) \qquad for\ 1<k<M \tag{3.4}$$

{where $t_{processing}(k)$ is the time in which processor P_k will complete processing jobs assigned to it.}

All the proposed algorithms, discussed in next sections, will try to maximize the fitness function by minimizing the *MS* and T_{delay}.

3.3 SexualGA based Resource Scheduling Algorithm (SGASchedule)

Genetic algorithms are a part of evolutionary computing, which is a rapidly growing area of artificial intelligence. Inspired by Darwin's theory about evolution, Genetic Algorithms (GAs) are adaptive heuristic search algorithm premised on the evolutionary ideas of natural selection i.e. survival of the fittest. Use of Genetic Algorithms (GAs) [82] in mapping jobs to resources, allows good solutions to be found quickly. So it allows the resource scheduler to be applied to more general problems. Many researchers have investigated the use of GAs to schedule tasks in homogeneous [82], [83] and heterogeneous [84]-[87] multi-processor systems with notable success. This section discusses the proposed SexualGA [89] based scheduling algorithm.

3.3.1 SexualGA

SexualGA [89] is an enhanced selection scheme that tries to mimic the sexual selection in human beings. In that context sexual selection is described as the concept of male vigor and female choice, meaning that male individuals try to spread their

gene material as wide as possible and female individuals are more selective by choosing rather above average fit males to guarantee a high survival probability of their off-springs. So it seems to be preferable not to use identical selection mechanisms for male and female population members also in GAs. Inspired by this view of sexual selection, a new selection mechanism for GAs (SexualGA) [89] is presented by introducing two different selection operators, one for the selection of male and one for the selection of female individuals.

3.3.2 Chromosome Representation

Schedule, *S,* of independent jobs is represented by a chromosome. Chromosome is basically a $N \times 1$ vector, where the position i ($0 < i < N$) represents the job / task and the entry at position i ($s[i]$) is the processor / machine to which the job has been assigned. For example, in Figure 3.1, $S\ [1] = P_2$, means job J_1 is assigned to Processor P_2.

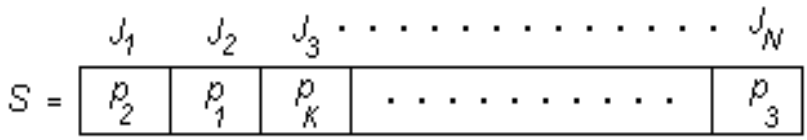

Figure 3.1: Chromosome representation

3.3.3 Initial Population

Initial population can be generated either randomly or by seeding. In the first method all the chromosomes are generated randomly from a uniform distribution where as in later some of the chromosomes in the initial population are generated using best-known techniques and others are generated using random generation. In the proposed algorithm seeding approach is used, where one chromosome is Min-min solution and rest is generated at random.

3.3.4 Selection

Genetic Algorithms use a selection mechanism to select individuals from population to insert into a mating pool. SexualGA [89] is new selection mechanism, which is used in proposed SexualGA based scheduling algorithm. The main idea of this scheme is to use two different selection schemes for selecting two parents required for each cross-over. The concept of male vigor and female choice are simulated using *Roulette Wheel Selection (RWS)* as the first selection scheme and *n-Tournament Selection* as the second selection scheme. So this selection mechanism has the advantage of not losing the genetic diversity and hence better control over the selection pressure.

3.3.5 Crossover and Mutation

Crossover operation is performed to generate new off-springs for the next generation. Two parents for the crossover are selected using the above-mentioned selection scheme and two-point crossover is performed. Using the vector representation for the chromosomes as shown in Figure 3.1, crossover operation is similar to exchanging the contents of two vectors for all positions before first crossover point and after the second crossover point. *Mutation* in the vector representation is considered as moving a job from one processor to another or swapping of two Jobs. For moving a job from one processor to another, two random numbers l and k are generated such that $S[l] = k$, where $0<l<N$ and $0<k<M$.

3.3.6 Exit criteria

An exit criterion specifies when to stop further exploration, and accept the solution. Proposed algorithm uses two exit criteria. 1) Standard deviation of the fitness value of the individuals in the population and 2) No change in the elite

chromosomes after 50 iterations. After the fitness function for all the chromosomes in the population is calculated, standard deviation is computed. If the standard deviation reaches a sufficiently low threshold value or there is no change in the elite chromosomes after 50 iterations, solution is said to converge.

3.3.7 Building the new population

New population is build from previous population through the crossover, mutation and elitism. This process is executed as follows. X_c % of the chromosomes, for the new population, is constructed using the crossover process as described earlier. X_m % of the chromosomes is constructed by applying the mutation operator. After the application of crossover and mutation, the rest of the chromosomes are taken out of the existing population by making them identical to existing population's best chromosomes, so as to make the size of new population equal to the existing one.

3.3.8 Algorithm (Pseudo Code)

```
// Initialize the variables
Initialize numberOfIterations
Initialize populationSize
Initialize Xc
Initialize Xm
Initialize threshHoldSD

// Array to hold elite chromosomes for population
EliteList eliteChromosomes[numberOfIterations]
// Generate initial population
pop0 = GenerateInitialPopulation()
// Calculate the fitness of individuals in population
pop0.CalculateFitness()
// Calculate Standard Deviation
pop0.CalculateSD()
done = false

// Evolve new generations
While (!done)
{
  // Generate next Population
  // perform crossover
  for ( i = 0 ; i < populationSize * Xc  ; i++)
```

```
  {
    // select first parent using Roulette wheel
    //selection
    parent1 = pop0.SelectParent(Type.RWS)
    // select 2nd parent using Tournament Selection
    parent2 = pop0.selectParent(Type.TOURNAMENT)
    // perform crossover and add to new population
    pop1.crossAndAdd(parent1, parent2)
  }

// perform mutation
  for ( i = 0 ; i < populationSize * Xm  ; i++)
  {
    pop1.mutateAndAdd(pop0)
  }
  // Add elite chromosomes
  pop1.addEliteChromosomes(pop0, eliteChromosomes)
  // Calculate fitness and standard deviation
  pop1.CalculateFitness()
  pop1.CalculateSD()

 // Check the criteria
  done = criteriaReached(pop1,
                   eliteChromosomes,
                   numberOfIterations,
                   threshHoldSD)
  pop0 = pop1
  numberOfIterations = numberOfIterations + 1
}
```

3.4 ACO based Resource Scheduling Algorithm (ACOSchedule)

Ant Colony Optimization (ACO) is a paradigm for designing meta-heuristic algorithms for combinatorial optimization problems. It is a probabilistic technique that was initially proposed by Marco Dorigo in 1992 in his Ph.D. thesis [90]-[93] to search for optimal path in the graph; based on the behavior of ants seeking a path between their colony and a source of food. Since then this idea was used to solve many compute intensive problems. In this section proposed grid scheduling algorithm based on ACO is discussed in detail. Many attempts have been made to apply ACO to scheduling problems [94]-[96]. Proposed algorithm differs from the other algorithms in two ways: 1) the objective function takes care of the interests of both, grid users

and resource providers as discussed in section 3.2.1 and 2) It uses the genetic operators to further optimize the results at the end, before the start of next iteration.

3.4.1 Pheromone Trail Definition

In ACO based algorithms ant's search is highly influenced by the pheromone trails, $\tau(i, j)$, left by other ants. So the definition of pheromone trail is one of the important decisions in ACO algorithms. Since grid is a heterogeneous environment, and some jobs may run more efficiently on some processors than others. So the pheromone will be stored in $N \times M$ matrix, which will have single entry for job processor pair in the problem. This value will represent the favourability of scheduling the job i on processor j.

3.4.2 Heuristic Information

One characteristic of a job that is very important is the critical time ($t_{critical}$) of a job. This is the minimum time in which a job can be processed if all the required resources are made available without any delay. Since the lower value is preferred so the inverse of this will be used.

$$\eta(i) = 1/(t_{critical}) \quad \{\text{where } t_{critical} \text{ is the critical time for job } i.\ \} \tag{3.5}$$

3.4.3 Pheromone Trail Updating

As mentioned above the pheromone matrix will have an entry for each job-processor pair in the problem, and each such pair in S_{best} 's solution (S_{best} is iteration's best solution) is reinforced in proportion to the relative fitness value of S_{best} compared to the global best solution, $S_{gb.}$ (S_{gb} is initially set to the Min-min solution). ρ is the rate of pheromone evaporation, where $0 \leq \rho \leq 1$. Hence

$$\tau(i,j) = \rho \cdot \tau(i,j) + F(S_{best}) / F(S_{gb}) \tag{3.6}$$

3.4.4 Building the solution

A specified number of ants are initialized and start building their tours in search of an optimal solution. The Job *j* is allocated to the processor *p*, in an arbitrary order, based on the pheromone value between *j* and *p* and critical time of *j*. The probability of selecting job *j* to be scheduled next is given by Equation 3.7.

$$Prob(j) = \frac{[\tau(j, p^{j}_{min})]^{\alpha} \cdot [\eta(j)]^{\beta}}{\sum^{n}_{i=1} [\tau(i, p^{i}_{min})]^{\alpha} \cdot [\eta(i)]^{\beta}} \quad (3.7)$$

where α and β are used to scale the pheromone trail and heuristic value respectively. If $\alpha = 0$, the decision is purely based on heuristic information and if $\beta = 0$ then the selection is purely on the basis of pheromone information for the job processor pair.

A job is then selected based on this value, and the chosen job *j* is then allocated to the processor, $p^{j}_{min,}$ on which it has minimum $t_{critical.}$ The schedule, found by an ant, at the end of iteration, is encoded in a vector *S*, of size *N*, where *N* is the number of jobs that are to be executed. *S[j]* indicates the machine where job *j* is assigned. This process is repeated until all jobs have been scheduled and a complete solution has been built. The pheromone trail update procedure is then used on the iteration best ant.

3.4.5 Optimizing ACO by Genetic Operators

At the end of iteration, when all the ants have built their tours, pheromone is updated by iteration's best ant. As the pheromone is deposited, the job processor pairs with more pheromone are more likely to be selected every time as compared to the other job processor pairs, with less pheromone value. So it may stop the further exploration of search space and solution may be limited to local minima. Hence, in

order to balance the load properly and to stop the solution to stuck to local minima, genetic operators are used.

The solution is represented in the form of chromosome by the same vector *S*, prepared by the ants while building their tours. The population size is equal to the number of ants. The solutions found by all the ants in the current iteration are the members of the population. The chromosomes are ordered in the non ascending order of the fitness of the solution. Crossover is performed selecting one parent from a specified percentage (*x* %) of chromosomes from lower end (solutions with lower fitness value) of the list and other parent from rest of the population. After performing few cross over operations a new population is prepared with *x* % new chromosomes and rest taken from the earlier population as it is. Again the fitness of all the solutions is taken and *SD* is computed. If there is any decrease in the *SD* for the population then the best ant from the new populations is used to update the pheromone trail other wise the iteration best ant, before applying the genetic operators is used to update the pheromone.

Mutation operator can also be applied. Mutation in the vector representation is considered as moving a Job from one machine to another or swapping of two Jobs. For moving a job from one machine to another, two random numbers l and k are generated such that $S[l] = k$, where $0<l<N$ and $0<k<M$. A limited number of crossover *(10 – 15)* or mutation operations (with least mutation probability) are performed so as not to loose the objective of finding an optimal solution in time.

Consider a simple example of mutation operator which optimizes the solution by decreasing the make span of the solution. Assume there are four jobs j_1, j_2, j_3, j_4 to be scheduled on processors p_1 and p_2. The completion time $t_{complete}$ of these jobs is as given in Table 3.1.

	j_1	j_2	j_3	j_4
p_1	3	4	2	3
p_2	4	3	4	1

Table 3.1: $t_{complete}$ of j_1, j_2, j_3, j_4 on $p_1 p_2$

Assuming that after the completion of the tour, the schedule S generated by an ant is represented by a vector as shown in Figure 3.2. Jobs j_1 and j_2 are to be processed on processor p_1 and jobs j_3 and j_4 are to be processed on p_2 as per this schedule, the makespan of the schedule is 7.

j_1	j_2	j_3	j_4
p_1	p_1	p_2	p_2

Figure 3.2: Schedule S before applying mutation operator
(Makespan = 7)

Now if the mutation operator is applied to this solution and a swap is performed in j_2 and j_3, the new schedule will be represented by vector S' as shown in Figure 3.3.

j_1	j_2	j_3	j_4
p_1	p_2	p_1	p_2

Figure 3.3: Schedule S' after applying mutation operator.
(Makespan = 5)

So the makespan for this schedule S', after the application of mutation operator, is 5 which is reduced as compared to previous solution formed by the application of pure ACO. Hence the use of genetic operators helps in exploring the complete search space and finding the optimal solution.

3.4.6 Algorithm (Pseudo Code)

```
// Initialize Variables

initialize NumberOfIterations
initialize NumberOfJobs
initialize NumberOfResources
initialize ResourceList[NumberOfResources]
initialize JobList[NumberOfJobs]
initialize MaxAnts
```

```
// Declare pheromone as 2d array
Initialize Pheromone[NumberOfJobs][NumberOfResources]

// Declare Solution pool to hold solution of all ants.
initialize SolutionPool[MaxAnts]

// input jobs and resource list
JobList = getJobsToSchedule()
RessoureList = getAvailableResources()

// Initialize global best solution as Min min solution
Initialize sGlobalBest[NumberOfJobs]
sGlobalBest = GenerateMinMinSchedule(JobList,
                                     ResourceList)

Repeat for every Ant
{
  while ( there are unscheduled Jobs in JobList )
  {
     For (every Resource in the resource list)
     {
     -  Get Next unscheduled Job.
     -  Compute probability to schedule on current
        resource.
     -  schedule the job on to the resource based on
        computed probability
     }

  }
  Add the Solution S to SolutionPool
}

// Find the iteration best solution in the solution pool
sIterationBest = findBestSolution (SolutionPool)

// Apply genetic operators find new iteration best
// solution
newSolutionPool = ApplyGeneticOpeartors (SolutionPool)
newIterationBest = findBestSolution(newSolutionPool)

if (computeFitness(sIterationBest) <
        computeFitness(newIterationBest))
{
   sIterationBest = newIterationBest;
}

// Update phreomone based on global best and iteration
// best solution
UpdatePheromone(Pheromone,
          sGlobalBest,
          sIterationBest);
```

```
// Update Global Best Solution if Required.
if(computeFitnesss(sIterationBest) >
            computeFitness(sGlobalBest))
{
   sGlobalBest = sIterationBest;
}

Repeat above steps for every iteration.

Output sGlobalBest as the final solution.
```

3.5 Hybrid Genetic Simulated Annealing based Grid Scheduling Algorithm (HybridGSA)

Genetic Simulated annealing heuristic is a hybrid of Genetic Algorithms and Simulated Annealing (SA) techniques. SA [97]-[102] is an iterative technique that considers only one possible solution at a time. SA uses a procedure that probabilistically allows poorer solutions to be accepted to attempt to obtain a better search of the solution space. This probability is based on a system temperature that decreases at every iteration. As the system temperature cools, it is more difficult for poorer solutions to be accepted. GA and SA both independently are valid approaches towards problem solving. However they have their own strengths and weaknesses. While GA can begin with a population of solutions in parallel, it has poor convergence properties. On the other hand SA has better convergence properties but cannot exploit parallelism.

Proposed HybridGSA algorithm provides a hybrid approach for the scheduling jobs on the grid. Hence HybridGSA algorithm maintains the diversity of GA while reducing the processing effort by following the temperature schedule as in SA. Since whole flow is same as SGASchedule algorithm described in section 3.3, so here we only describe parts that are different from SGASchedule.

3.5.1 Initial Temperature and Annealing Schedule

Initial temperature, *T*, is set to a value such that the acceptance probability for the chromosomes of the population is very high i.e. close to *1.0*. The annealing schedule is generated by multiplying the current temperature by a constant less than *0*, which is known as the cooling rate, Υ. The probability of accepting new solution in reproduction (crossover and mutation) is implemented as a function of temperature, *T*. So initially the new off-springs are accepted with a high probability, where as, in the later stage their probability of being accepted is lowered as the temperature decreases. Temperature, *T,* is decreased at cooling rate, Υ, after every iteration and is given by Equation 3.8.

$$T_i = \Upsilon * T_{i-1} \quad \{\text{where } i \text{ is the iteration number}\} \tag{3.8}$$

3.5.2 Crossover

Crossover operation is performed to generate new off-springs for the next generation. Two parents for the crossover are selected using the above-mentioned selection scheme and two-point crossover is performed. Using the vector representation for the chromosomes as shown in Figure 3.1, crossover operation is similar to exchanging the contents of two vectors for all positions before first crossover point and after the second crossover point. After the generation of new off-springs, change in the energy, δE, is computed using the Equation 3.9 and their probability, *P*, of acceptance is given by Equation 3.10.

$$\delta E = f_{offspring} - f_{Parent} \tag{3.9}$$

where $f_{parent} = Max\ (f_{Parent1}, f_{Parent2})$, Maximum fitness of two parents

$f_{offspring}$ = Fitness of the new offspring

$$P = exp\ (\ -\delta E / T)\quad \{P = 1\ if\ \delta E > 0\} \tag{3.10}$$

After the finding the probability of acceptance, a uniform random number, μ, is generated between *0* and *1*. New offspring is accepted if, $P >= \mu$, otherwise the new offspring is rejected.

3.5.3 Mutation

Mutation in the vector representation is considered as moving a job from one resource to another or swapping of two Jobs. For moving a job from one resource to another, two random numbers *l* and *k* are generated such that *S[l] = k,* where $0<l<N$ and $0<k<M$. After applying the mutation, the change in energy, δE, is computed using Equation 3.9 and probability of acceptance, *P*, is computed using Equation 3. 10. If, $P > \mu$, (a uniform random number between *0* and *1*), the new offspring is added to the next population, otherwise it is rejected.

3.5.4 Algorithm (Pseudo Code)

```
// Initialize the variables
Initialize numberOfIterations
Initialize populationSize
Initialize Xc
Initialize Xm
Initialize threshHoldSD
Initialize SystemTemprature
Initialize CoolingRate
// Array to hold elite chromosomes for population
EliteList eliteChromosomes[numberOfIterations]
// Generate initial population
pop0 = GenerateInitialPopulation()
// Calculate the fitness of individuals in population
pop0.CalculateFitness()
// Calculate Standard Deviation
pop0.CalculateSD()
done = false
// Evolve new generations
While (!done)
{
```

```
  // Generate next Population
  // perform crossover
  for ( i = 0 ; i < populationSize * Xc  ; i++)
  {
    // select first parent using Roulette wheel
    //selection
    parent1 = pop0.SelectParent(Type.RWS)
    // select 2nd parent using Tournament Selection
    parent2 = pop0.selectParent(Type.TOURNAMENT)
    // perform crossover and add to new population
     offsprings[2] = parent1.cross(parent2);
     // Find out the max fitness of two parents
     fParent = Max(parent1.fitness(), parent2.fitness());
     // Check if first offspring is to be added to new
     // population
     DeltaE = offspring[0].fitness() - fParent;

     if (DeltaE > 0 ) P = 1;
     else  P = exp ( -DeltaE/T);

     if ( P > Random.Getdouble() )
     {
          pop1.Add(offspring[0]);
     }
     // Check if second offspring is to be added to new
// population
     DeltaE = offspring[1].fitness - fParent;
     if (DeltaE > 0 ) P = 1;
     else P = exp (-DeltaE/T);

     if ( P > Random.Getdouble() )
     {
          pop1.Add(offspring[1]);
     }

}

// perform mutation
  for ( i = 0 ; i < populationSize * Xm  ; i++)
  {
     offspring = parent1.mutate();
     DeltaE= offspring.fitness() - parent1.fitness();
     if (DeltaE > 0 ) P = 1;
     else P = exp (-DeltaE/T);

     if ( P > Random.Getdouble() )
     {
          pop1.Add(offspring);
     }
  }
```

```
  // Add elite chromosomes
  pop1.addEliteChromosomes(pop0, eliteChromosomes)
  // Calculate fitness and standard deviation
  pop1.CalculateFitness()
  pop1.CalculateSD()
  // Check the criteria
  done = criteriaReached(pop1,
                   eliteChromosomes,
                   numberOfIterations,
                   threshHoldSD)
  pop0 = pop1
  numberOfIterations = numberOfIterations + 1

  // Lower the temperature
  SystemTemprature = CoolingRate * SystemTemprature;
}
```

3.6 Summary

This chapter starts with the identification of Quality of Service (Qos) parameters and then formulates an objective function based on these QoS parameters. This objective function tries to take care of interests of users by minimizing the cumulative time delay and also tries the maximum utilization of resources by minimizing the make span of the schedule. Further three new scheduling algorithms, SGASchedule, ACOSchedule and HybridGSA are proposed and discussed in detail. Next chapter provides a detailed discussion on the proposed ACO guided mobile agents based resource discovery algorithm.