CHAPTER 6

HEURISTIC APPROACHES FOR NON-DIVISIBLE JOBS WITH PRECEDENCE CONSTRAINTS

In this chapter, the aim is to find an energy-efficient schedule for a given system that also satisfies the precedence constraints between jobs to be performed by the system. The machines in the system considered in this chapter have identical speed and the jobs are non-divisible. It has been established that scheduling of jobs on a multi-machine system to reduce the makespan, while satisfying the precedence constraints between the jobs, is an NP-hard problem [42]. In Chapter 3 Section 3.4, we have seen that the energy efficient scheduling is a generalization of makespan scheduling. Clearly, the present class of energy efficient scheduling problems is also NP-hard. As we have mentioned in Chapter 2, heuristic and search based approaches have performed well for makespan scheduling with the current system model under consideration.

We propose three different algorithms for energy-efficient scheduling and indicate their strengths and limitations. The current system model is very generic and can describe many real world distributed systems from different domains. We have validated our algorithm with simulations of systems with different numbers of machines, with some standard program graphs. Each of the proposed algorithms has their own strengths and limitations. In the next sections we explain these methods in detail. We first explain the genetic algorithm based method.
6.1 Genetic Algorithm for EES

A genetic algorithm [108–110] (GA) is a heuristic for search, mimicking the natural evolution process. In a genetic algorithm, a population of strings encoding candidate solution evolves towards better solutions using mutation and crossover functions. The evolution usually starts from a random population of individuals and happens over multiple generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations of candidate solutions are completed, or a satisfactory fitness level is reached for the population.

Genetic algorithms have been used for makespan scheduling with good success [111, 112]. We present a method which instead uses this power of GAs to find energy-reducing schedules.

We first map the energy-efficient scheduling problem to the GA domain. Since a GA finds the strings which have good fitness values, we model schedules as strings and the energy values of the strings by the fitness function. Hence a string in GA is a vector of length equal to the number of nodes in the program graph. Each element in that vector represents the machine corresponding to that node. The parents for mutation and crossover are chosen on the basis of the energies of the schedules in the current generation. A schedule is more likely to be chosen as a parent if its rank (in terms of lower energy consumption) is better than others. The mutation rule uses a Gaussian distribution with zero mean and a variance, which reduces the number of generations. The crossover is random, in the sense that it arbitrarily chooses the portions from the first and second parent. The search space of a GA depends upon the number of machines
as well as the number of elementary jobs in the program graph. The total number of possible solutions in a GA-based method is $m^n$.

In the next section we explain our second proposed algorithm which uses a cellular automaton and a genetic algorithm to find energy-efficient schedules.

### 6.2 Cellular Automata Based Method for EES

We use the framework of cellular automata (CA) to solve the stated scheduling problem. But to solve the scheduling problem using CA, the system graph and program graph have to be first mapped to the CA domain. To understand how we have done so, a little background of CA is provided first.

#### 6.2.1 Background: Cellular Automata

Cellular automata are groups of cells with well-defined structures and behavior. These cells can have discrete states defined at discrete time instances. The state of a cell at a particular time instant in future depends upon its own state as well as its neighboring cells’ states at the present time instant. A set of rules specify the future state of any cell according to the present state of that cell and its neighborhood. These set of rules are same for all the cells. The neighborhood depends upon the structure of the automata. Generally the neighborhood of a cell also contains the cell itself, hence we can generally say that the next state of a cell depends upon the current state of its neighborhood.

Figure 6.1 shows a linear cellular automaton in which cells are places next to one another forming a one-dimensional array. In this automaton, a cell can have discrete integer states from 1 to 8.
Figure 6.1: One dimensional cellular automata. The numbers inside the cells represent their states.

The size of a CA neighborhood is generally specified by the radius of the neighborhood. If the radius is 2 in a regular linear CA, then the neighborhood consists of the cell itself and its left two and right two adjacent cells. Hence the size of the neighborhood is then equal to 5. A two-dimensional cellular automaton has cells arranged in a rectangular grid pattern, as shown in Figures 6.2 and 6.3. If the radius of the neighborhood is specified as 1, the neighborhood may consist of the 5 cells (as shown by dark cells in Figure 6.2) or it may contain 9 cells (as shown by dark cells in Figure 6.3). Hence it is the modeler’s choice to decide the neighborhood. The selection of a suitable neighborhood is often critical for the model to be good.

Figure 6.2: Two dimensional cellular automata with neighborhood of size 5

The neighborhood is not always formed just according to geometric proximity of
the cells but is selected by a certain method. Such cellular automata are called *irregular* cellular automata since cells that are not in geometric proximity to a given cell may also be included in its neighborhood. The calculation of the neighborhood for cells at the edges may also use a different method than for cells in the middle of the automaton array, since edge cells may not have appropriate neighbors to apply a general method.

The *rule* for describing the transition of a cell’s state based on its neighborhood is also a main modeling aspect. A rule is generally specified by a table which contains all the possible neighborhood states and the corresponding future state of the cell. An example rule is shown in Table 6.1.

<table>
<thead>
<tr>
<th>State of Neighborhood</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future state of Cell</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.1: An example rule table with neighborhood size 3 and number of possible states is two. States are given by 0 and 1.

An elementary job of the program is mapped to a cell in the CA space. The state
of the cell specifies the machine to which the job is assigned. Initially, a random assignment of jobs is created. Then according to the rules and the neighborhood, the CA evolves successively to reach a state which gives a better schedule. We use a genetic algorithm (GA) to find the CA rules which in turn generate better schedules. In the process of finding better rules, we also keep improving on the schedules across GA generations.

The detailed architecture of this method is explained in Section 6.2.2. The algorithm for energy-efficient scheduling is described in Section 6.2.3. The selection of the neighborhood and the rules are critical to finding a good solution with minimal scheduling overhead. Section 6.2.4 describes the cellular automaton algorithm and the method for selecting the neighborhood. The learning of rules and the finding of a good schedule are explained in Section 6.2.5.

6.2.2 Architecture

There are four major building blocks in our architecture. The block diagram in Figure 6.4 presents the architecture of our method. The four blocks named initializer, CA module, genetic algorithm and analyzer, represented by dotted lines, are as described below.

- **INITIALIZER**: This takes the program graph and the system graph as inputs. A MAPPER in the INITIALIZER maps the system graph with the program graph, which enables the INITIALIZER to generate $s$ initial random schedules and also $r$ initial random rules for the CA MODULE. When the algorithm starts for the first time, these schedules and rules are used to initialize the current set of schedules and the current set of rules respectively.
Figure 6.4: Schematic for the CA+GA scheduling method
CA MODULE: The inputs to the CA MODULE are schedules and rules, which are taken from the current schedules and current population of rules respectively. The CA algorithm evolves upon the current schedules using the current set of rules. There are \( s \) schedules, each of which is acted on by \( r \) rules, so there are \( s \times r \) output schedules. (For details of how the cellular automaton evolves, refer Section 6.2.4.)

GENETIC ALGORITHM: The work of the genetic algorithm (GA) is to search for good rules in the space of all rules. It takes rules and energy values of schedules as inputs, and based on these energy values it selects elite rules and applies mutation and crossover functions to the rest of the rules and produces a better generation of rules. For the next iteration these elite rules and the rules generated by GA are used to populate the current set of rules. Section 6.2.5 explains the further details of our genetic algorithm.

ANALYZER: The output of the CA module, i.e., the set of \( s \times r \) schedules, is taken as input by the analyzer, and for each schedule the energy is calculated, and schedules are sorted by their energy values. The best \( s \) schedules are given to the CA module for the next iteration, and the best \( r \) rules are given to the genetic algorithm. Also, the best schedule \( (\sigma^*) \) is taken as output from this block when the stopping criterion for the GA is reached.

6.2.3 Algorithm

Notation used in Algorithm 5:

- \( s \) is the total number of initial schedules
- \( r \) is the total number of initial rules
- \( \sigma_m \) is the \( m^{th} \) schedule
• $\rho_n$ is the $n^{th}$ rule

• $\xi()$ is the energy function for schedules

• $\xi(\sigma_m)$ is the energy consumed by the $m^{th}$ schedule

• $\nu$ is the current generation

• $g$ is maximum number of generations

• $\varsigma_{m,n}$ is a schedule got after applying CA to the $m^{th}$ schedule with the $n^{th}$ rule

• $\sigma^*$ is the schedule with minimum energy

• $\rho_n$ is the updated rule after applying GA to rule $\rho_n$

Algorithm 5 indicates the CA+GA algorithm, and may be read in correlation with our architecture described previously.

6.2.4 Cellular Automaton Module

The architecture of the CA used in the proposed method is linear and irregular, which means that the neighborhood of a cell need not necessarily consist of the geometric neighbors of the cell. This provides the scope to choose neighbors that are more relevant according to the program graph. Our proposed method uses a neighborhood of size 4, which includes two parents (nodes in a program graph) and two children (nodes in a program graph) of the job. If a node $i$ has more than two parents (respectively, children) then 2 parents (respectively, children) are selected from all available parents of the node. If a node has less than 2 parents (respectively, children) then dummy nodes are assigned to it as neighbors. The process of assigning nodes as neighbors is explained below:
Algorithm 5: The CA+GA Algorithm

input : Initial schedules \((\sigma_m)\), rules \((\rho_n)\)

output: Best schedule \((\sigma^*)\)

1 \(\rho_n \leftarrow \rho_n, 1 \leq n \leq r;\)
2 \(\sigma_m \leftarrow \sigma_m, 1 \leq m \leq s;\)
3 \(\xi(\sigma_m) \leftarrow 0;\)
4 heapsort\((\xi(\sigma_m))\);
5 \(\xi(\sigma_q) \leftarrow \min(\xi(\sigma_m));\)
6 \(\sigma^* \leftarrow \sigma_q;\)
7 \(\nu \leftarrow 1;\)
8 for \(n = 1\) to \(r\) do
   9    for \(m = 1\) to \(s\) do
10       \(\varsigma_m,n \leftarrow CA(\rho_n, \sigma_m);\)
11       \(\xi(\varsigma_m,n);\)
12    end
13 heapsort\((\xi(\varsigma_m,n))\);
14 \(\sigma_m \leftarrow \varsigma_m,n, 1 \leq m \leq s;\)
15 if \((\nu = g) || (\min(\xi(\varsigma_m,n)) \text{ has not decreased } d \text{ consecutive times})\) then
16       heapsort\((\xi(\sigma_m))\);
17       \(\xi(\sigma^*_v) \leftarrow \min(\xi(\sigma_m));\)
18       if \(\xi(\sigma^*_v) < \xi(\sigma^*)\) then
19          \(\sigma^* \leftarrow \sigma^*_v;\)
20    end
21 else
22       \(\rho_n \leftarrow GA(\rho_n);\)
23       \(\nu \leftarrow \nu + 1;\)
24       \(\rho_n \leftarrow \rho_n, 1 \leq n \leq r;\)
25 goto step 6;
26 end
1. Two dummy parent nodes with state -1 are assigned to the entry nodes (i.e., nodes with no parent node).

2. Two dummy children nodes with state -1 are assigned to the exit nodes (i.e., nodes with no child node).

3. If a node has only one parent then a dummy parent node is added with the same state as the non-dummy parent node.

4. If a node has only one child then a dummy child node is added with the same state as the non-dummy child node.

5. If a node $i$ has more than two parents (respectively, children) then the two parents (respectively, children) connected with highest edge weights ($l(j,i)$) to the node $i$ are selected as the parents (respectively, children).

We choose the parents (respectively, the children) with highest edge weights in the CA neighborhood, since these edges affect the objective function more than others. Once the scheduling problem is mapped to the cellular automata domain, we proceed to find good schedules. Such good schedules are given by the evolved state of the automaton, which is obtained by applying elite rules to good initial schedules. Since both the rules and the schedules are crucial, we search for both of these. We have designed a procedure in which we keep improving the state of the CA as well as the rules simultaneously to find the near-optimum schedule. We use a genetic algorithm to search for the good rules. The process is explained in the next Section.

### 6.2.5 Genetic Algorithm for Rule Selection

We use a genetic algorithm (GA) to search for good rules and in the process of doing so we also find the near-optimal schedule for our scheduling problem. The application of genetic algorithms to finding CA rules was first discussed by Das et al. [113]. As
the aim is to search for good CA update rules, they are treated as individuals, creatures, or phenotypes in our GA setting. The fitness of a rule is given by the energy efficiency of the schedules obtained by applying the rule to some initial states of automata. Since rules can be represented as number strings, the reproduction is carried out by mutation and crossover of these strings.

The parents for mutation and crossover are chosen on the basis of average energies of the rules in the current generation. A rule is more likely to be chosen as a parent if its rank (in terms of lesser energy consumption) is better than others. The mutation rule uses a Gaussian distribution with zero mean and a variance which reduces over generations. The crossover is also random in the sense that it arbitrarily chooses the portions from first and second parents.

In the next Section we explain the heuristic-based approach to energy-efficient scheduling.

6.3 Efficiency-Based Allocation Heuristic (EAH)

The two methods we have till now are search algorithms. Search algorithms just try to find the best possible solution in a given solution space. The other types of algorithms generally used for scheduling are heuristic-based algorithms. Such algorithms have some heuristic at their core. The heuristic proceed towards locally optimal results and it is hoped that with the combinations of locally optimal decisions, a globally optimal or at least a good result can be obtained. Heuristic-based algorithms are very fast and have steady performances, but generally reach some sub-par solution. Search-based algorithms are slower but on the other hand do not get stuck in local minima, though they do not guarantee a good solution every time they are executed. Their success can only be measured statistically.
The First-In-First-Out (FIFO) heuristic is used for scheduling in many single-machine as well as multi-machine systems. We have extended this algorithm by using the knowledge of different power requirement of different machines. As soon as a job is ready to be executed, we check which machines have an empty input queue. From the set of free machines we choose the one with lowest working power, and we assign the job to that machine. If no machine is free, then we assign the job to the machine with least working power.

We now present the simulation experiments and their results which substantiate the usefulness of our proposed methods.

6.4 Results

A number of simulations with standard program graphs have been conducted. These graphs are tree15, g18, gauss18 and g40 which are also used extensively in the literature (see, e.g., [42]).

Our algorithms as well as the simulation setup allow for any number of machines, which can have different working state power consumptions and idle state power consumptions, to be used in the simulated systems. However, most existing scheduling algorithms consider fewer than 8-machine systems. The standard graphs which we have used are tested with 8-machine or smaller systems in previous published work. Hence for illustrative purposes we have shown results for 2-, 4-, and 8-machine systems, to make it easier to compare our algorithms with others. Similar results also obtain with other numbers of machines.

As has been previously discussed, prior state-of-the-art scheduling algorithms compute good schedules for minimizing makespan rather than energy. Though our algorithms are energy-efficient and work to reduce the energy rather than time, if we take the
working power and idle power as identical then they provide the minimum makespan schedules as well. Hence for the sake of comparison with other systems we have also calculated the schedules for makespan as well.

In the simulations reported in this Section for the CA+GA algorithm (Algorithm 5), we assume that the cellular automaton works asynchronously [42]. This means at a given instant of time, only one cell updates its state. So, a single step of the CA, i.e., a rule to be applied once on all cells, takes on the order of the number of jobs to be completed. In the simulation, for learning the rules we fixed the population size of the GA at 20 and the maximum number of generations at 100. These tuning parameters are kept the same for all approaches presented.

For the GA scheduling algorithm also we kept the GA parameters the same.

The schedules obtained from the CA+GA and GA algorithms just tell us which job will be performed on which machine, but not the sequence of job execution. We have used a FIFO approach for scheduling of jobs assigned to the same machine, i.e., the job which reaches the machine earlier is executed first.

In the following Section we first explain the different program graphs which we have used. We then analyse different aspects of our proposed methods based on the simulation results. After this we present the comparison of our methods with one another and also with other published methods. In the end, we discuss trade-offs and indicate the advantages of our proposed algorithms based on experiments.

6.4.1 Program Graphs

We used four standard program graphs for simulation experiments: tree15, g18, gauss18 and g40. Tree15 is a binary tree with 15 nodes. All the working costs and communication costs in this program graph are the same, and can be taken to be unity.
The program graph for graph tree15 is shown in Figure 6.5.

![Program graph tree15](image)

Figure 6.5: Program graph tree15

The program graph g18 is displayed in Figure 6.6. It has 18 jobs with different computation costs as shown in the figure, and the communication cost for all links equal unity.

Figure 6.7 displays the program graph g40 which is also considered in simulation. This is a directed acyclic graph which has 40 nodes. The computation and communication costs of jobs are equal to 4 and 1 respectively.

Figure 6.8 displays the program graph gauss18. This is a directed acyclic graph which has 18 nodes. The computation and communication costs of jobs are as indicated in the figure.

6.4.2 System Analysis: Effects of Varying $z$ and $m$

In this section we present our analysis of our experiments in which we have simulated the program graphs on different system graphs. These system graphs have different number of machines ($m = 2, 4, 8$) and different idle power to working power ratios
Figure 6.6: Program graph g18
Figure 6.7: Program graph g40

(z = 0, 0.1, 0.5, 1). The energy and makespan values shown in this section are the best ones we have got from our energy-efficient algorithms. The focus of the analysis is how \( z \) and \( m \) are related irrespective of the scheduling algorithm used. The comparison of energy values with different values of \( z \) for 2, 4 and 8 machines for program graph tree15 and g40 is shown in Figure 6.9 and 6.10 respectively.

We see that with an increase in the number of machines, the energy required also increases if jobs are less—because in that case most of the machines remain idle and the idle power consumption contributes to increase in the total energy \( E \), though the makespan time is decreased because jobs are executed in parallel on different machines.

The effect of any increase in idle power consumption of machines on the net system energy consumption is pretty straight-forward. As the idle power consumption of machines increases, the energy consumption of the system increases nearly linearly for a
Figure 6.8: Program graph gauss18
Figure 6.9: Comparison of energy values with different idle times for 2, 4 and 8 number of machines for program graph tree15

Figure 6.10: Comparison of energy values with different idle times for 2, 4 and 8 number of machines for program graph g40
system with high number of machines as compared to parallelization capability of the program graph. Figure 6.11 shows the variation in energy consumption with variation in $z$ for a 4 machine system on g40 graph.

These plots clearly show that as the number of machines increase, the idle power specification plays an important role in deciding the energy consumption of the schedule.

### 6.4.3 Comparative Performance of Algorithms

As there is relatively less literature on energy-efficient scheduling *per se*, we first compare our scheduling algorithms with other algorithms which optimize on makespan. As previously noted, the energy-efficient scheduling becomes makespan scheduling if all machines have the same power specification.

Using a two-machine system, we show the best makespan calculated by our proposed as well as other algorithms along with the time taken to calculate the schedules in Table 6.2, similar relations are seen for other number of machines as well. Each entry
in this table has two elements separated by a comma. The first element indicates the makespan time and the second represents the scheduling overhead time of the algorithm in seconds.

<table>
<thead>
<tr>
<th>Graphs</th>
<th>CA+GA</th>
<th>Plain GA</th>
<th>EAH</th>
<th>Greedy-Min</th>
<th>Greedy-Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree15</td>
<td>9, 3.76</td>
<td>9, 0.32</td>
<td>12, 0.002</td>
<td>12, 0.015</td>
<td>12, 0.01</td>
</tr>
<tr>
<td>g18</td>
<td>46, 4.97</td>
<td>46, 0.43</td>
<td>58, 0.005</td>
<td>76, 0.05</td>
<td>76, 0.04</td>
</tr>
<tr>
<td>gauss18</td>
<td>44, 6.53</td>
<td>44, 0.66</td>
<td>102, 0.005</td>
<td>74, 0.035</td>
<td>74, 0.03</td>
</tr>
<tr>
<td>g40</td>
<td>80, 20.26</td>
<td>80, 0.96</td>
<td>112, 0.01</td>
<td>89, 0.197</td>
<td>89, 0.11</td>
</tr>
</tbody>
</table>

Table 6.2: Makespan comparison

We used an Intel i5 processor computer for these simulations. In Tables 6.2, 6.3 and 6.4, EAH refers to the efficiency based allocation heuristic explained in Section 6.3. In the same table, GA refers to the genetic algorithm based energy-efficient scheduling algorithm explained in Section 6.1. The greedy-min algorithm is a standard algorithm [86] which tries to first finish the jobs which can be finished sooner (i.e., the jobs with lower node weights) while honoring the program graph dependencies. In greedy-max algorithm we give preference to finish the jobs first which take long time to finish (i.e., the jobs with higher node weights) while adhering to the program graph dependencies.

Clearly the CA+GA algorithm gives the best results for makespan scheduling. We have not mentioned the results of other state-of-art algorithms in the table since the makespan values provided by them are the best possible for these graphs, and we have also got the same makespans. This indicates that our algorithms perform as good as other good published algorithms for makespan scheduling. We present another table below which shows the energy requirements of all the algorithms considered.

The different energies comparison of all the methods on a 4-machine system is shown in Table 6.3 and that for a 8 machine system is given in Table 6.4.

As suggested by the relative dearth of learning algorithms in the field of scheduling,
Table 6.3: Energy comparison, with $m = 4$

<table>
<thead>
<tr>
<th>Graphs</th>
<th>CA+GA</th>
<th>Plain GA</th>
<th>EAH</th>
<th>Greedy-Min</th>
<th>Greedy-Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree15</td>
<td>36.4, 5.02</td>
<td>36.8, 0.428</td>
<td>38.8, 0.015</td>
<td>41.6, 0.063</td>
<td>41.6, 0.028</td>
</tr>
<tr>
<td>g18</td>
<td>201.2, 7.58</td>
<td>201.6, 0.31</td>
<td>223.2, 0.004</td>
<td>311.6, 0.048</td>
<td>311.6, 0.038</td>
</tr>
<tr>
<td>gauss18</td>
<td>163.2, 8.735</td>
<td>193.6, 0.359</td>
<td>291.2, 0.003</td>
<td>311.6, 0.029</td>
<td>311.6, 0.029</td>
</tr>
<tr>
<td>g40</td>
<td>356, 19.5</td>
<td>374.4, 1.007</td>
<td>406.4, 0.009</td>
<td>436.8, 0.134</td>
<td>436.8, 0.14</td>
</tr>
</tbody>
</table>

Table 6.4: Energy comparison, with $m = 8$

<table>
<thead>
<tr>
<th>Graphs</th>
<th>CA+GA</th>
<th>Plain GA</th>
<th>EAH</th>
<th>Greedy-Min</th>
<th>Greedy-Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree15</td>
<td>76.8, 87.76</td>
<td>84, 0.71</td>
<td>76.8, 0.013</td>
<td>104, 0.048</td>
<td>104, 0.028</td>
</tr>
<tr>
<td>g18</td>
<td>380.8, 96.33</td>
<td>412.8, 0.36</td>
<td>380.8, 0.006</td>
<td>652.8, 0.102</td>
<td>652.8, 0.07</td>
</tr>
<tr>
<td>gauss18</td>
<td>454.4, 62.11</td>
<td>462.4, 1.23</td>
<td>834.4, 0.01</td>
<td>820.8, 0.086</td>
<td>820.8, 0.104</td>
</tr>
<tr>
<td>g40</td>
<td>687.2, 204.2</td>
<td>761.6, 0.8288</td>
<td>786.4, 0.008</td>
<td>931.2, 0.372</td>
<td>931.2, 0.398</td>
</tr>
</tbody>
</table>

it is not straightforward to apply learning algorithms to scheduling. Learning algorithms can learn to derive good schedules but such learning, and the schedules obtained, are very specific to particular program graphs. It is generally better to store the good schedules for each program graph rather than trying to learn them, hence learning schedules is not a good option. With CA+GA we do not learn schedules, but use the GA to learn CA rules that generate good schedules. These CA rules are specific to the system graph but not to the program graph (the same rules can be used to generate schedules for different program graphs on a given system), i.e., they can in a sense capture the properties of the system graph.

Among our proposed algorithms, CA+GA and Plain GA are search algorithms, hence can give different outputs on different runs, while EAH (like the published greedy-min and greedy-max) is a heuristic-based algorithm which gives the same output each time. The output of search algorithms is better than heuristic-based algorithms which shows that simple heuristics are not good enough to find the best schedules and they always come to some locally optimum results. Hence search algorithms are necessary to find better results. Clearly as can be seen in the results, the Plain GA algorithm is as good as the CA+GA algorithm and it is much faster. But there are certain advantages
of the CA+GA algorithm over others:

- **Better Convergence**: In our experiments we have seen that CA+GA gives very good results much more frequently than the Plain GA.

- **Fixed Search Space**: The search space of the CA+GA algorithm only depends on the number of machines and the size of the neighborhood of the CA. On the contrary, the search space of GA is exponentially proportional to the number of nodes of the program graph. The bigger the search space, the lower the probability of reaching the global optimum.

- **Learning**: The biggest motivation behind using CA is its ability to learn the characteristics of the system graph. A rule learned for one program graph on a given system can be applied successfully to other program graphs on the same system.

We conducted experiments which support this claim of learning ability of CA. We obtained a CA rule learned for a given system and program graph, which we then applied on random schedules of other program graphs to get the evolved schedules. We noted that the evolved schedules were on an average much better than the initial random schedules. The percentage improvement for all the graphs is reported in Table 6.5. The percentage improvement is calculated as:

\[
impr = \frac{1}{I} \sum_{i=1}^{I} \frac{E_i(\text{initial}) - E_i(\text{evolved})}{E_i(\text{initial})} \times 100 \tag{6.1}
\]

Here \( I \) is the number of initial random schedules which we have taken as 200 in our experiments. The first column of Table 6.5 indicates the program graph on which the rule was learned. Each row records the percentage improvement in the schedule computed using a rule learned with the program graph mentioned in the first column of that row, over the initial (random) schedule. We consider a 4-machine system with power consumptions 1, 2, 3 and 4 respectively, with \( z \) taken as 0.2.
As noted previously, a CA rule learned with a program graph can be gainfully applied to another program graph. For instance, we see from Table 6.5 that the rule learned using program graph g18, when applied to program graph gauss18, gives a 50.55% improvement.

<table>
<thead>
<tr>
<th>Graphs</th>
<th>tree15</th>
<th>g18</th>
<th>gauss18</th>
<th>g40</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree15</td>
<td>34.95</td>
<td>32.19</td>
<td>52.78</td>
<td>28.63</td>
</tr>
<tr>
<td>g18</td>
<td>33.47</td>
<td>31.98</td>
<td>50.55</td>
<td>29.03</td>
</tr>
<tr>
<td>gauss18</td>
<td>35.42</td>
<td>30.52</td>
<td>49.02</td>
<td>27.61</td>
</tr>
<tr>
<td>g40</td>
<td>29.43</td>
<td>22.72</td>
<td>37.32</td>
<td>28.03</td>
</tr>
</tbody>
</table>

Table 6.5: Learning simulation

The learning capability of CA indicates that the algorithm using CA+GA is not just a random global search for better rules, but constitutes searching in a reasonable manner. Hence the CA+GA algorithm has the power of learning which helps it converge to better solutions, while remaining a search algorithm which ensures that we do not get stuck into locally good results.

Hence all the proposed algorithms have certain advantages over others and we would sum up our recommendations based upon the properties of the algorithms:

- **EAH**: It is a very simple and fast algorithm and needs little memory and computational power compared with other approaches. It is to be preferred if getting the best possible schedule is not a high priority but the schedule is to be calculated fast (or at run-time), or if the system calculating the schedule has limited processing power.

- **Plain GA**: It is a fast search algorithm. It is to be preferred if sufficient processing power is available and good schedules, though not necessarily the best ones, are to be calculated quickly.

- **CA+GA**: It gives the best schedules, and is to be preferred if the scheduling overhead can be overlooked (as in industrial settings) in order to obtain schedules that
are likely to be the best obtainable.

Hence a user or designer can choose from any of the proposed algorithms based on the requirements. We find the CA+GA algorithm to give the best schedules amongst the proposed algorithms. The next Section presents some more insights in the CA+GA algorithm using some simulation data.

6.4.4 Insights into the CA+GA Algorithm

The proposed CA+GA algorithm is quite complex. Looking at some graphs giving simulation results can help us understand more about the algorithm. We analysed the change in the fitness function for GA across different generations. The fitness function of a rule is the average of energies of the output schedules which are evolved using that rule. The plot for graph treel5 is shown in Figure 6.12, which shows the best, worst and mean values of the fitness function (energy function) of different rules across the generations. Similar graphs for g18 and g40 are shown in Figures 6.13 and 6.14. In case of g18 and g40, the mean values of fitness function are decreasing with generations, which shows that better rules have been found as the generations are progressing.

On the contrary, in Figure 6.12 it is seen that the minimum value of fitness function were better in earlier generations rather than later generations. This can be explained from the fact that the input schedules on which the rules are tested are different in different generations, hence even if the elite rules of previous generations are carried forward to the next generation, its fitness value may have increased. Though this gives us an indication that we might not be improving upon the rules over generations, but it does not affect adversely our final result. The reason is that we store the best schedules from across generations and give the one with lowest energy as the output schedule regardless of the generation in which that schedule was evolved. In the-above mentioned simulation results, we have chosen the value of $z$ to be 0.1, which means that the idle
power of the machines is taken as 0.1 times their working power.

Figure 6.12: Plot for program graph tree15 showing the best, worst and mean values of the fitness function across generations

In this chapter we have presented three algorithms for energy-efficient scheduling of jobs on a system with multiple, non-identical machines that have non-zero idle power requirements. Our approach is quite generic, in the sense that it can be used to model and analyse many systems in various domains. All three proposed algorithms give results comparable to or better than present state-of-the-art methods.
Figure 6.13: Plot for program graph g18 showing the best, worst and mean values of the fitness function across generations

Figure 6.14: Plot for program graph g40 showing the best, worst and mean values of the fitness function across generations