INT. J. ELECTRONICS, 1994, VOL. 77, NO. 4, 401–415

BENTEN MST, SADIQ M. SAIT•GENETIC SCHEDULING OF TASK GRAPHS •INTERNATIONA
JOURNAL OF ELECTRONICS 77 (4): 401-415 OCT 1994 •

Genetic scheduling of task graphs

MUHAMMAD S. T. BENTEN† and SADIQ M. SAIT†

A genetic algorithm for scheduling computational task graphs is presented. The problem of assigning tasks to processing elements as a combinatorial optimization is formulated, and a heuristic based on a genetic algorithm is presented. The objective function to be minimized is the 'time on completion' of all tasks. Results are compared with those published in the literature and with randomly generated task graphs whose optimal schedules are known *a priori*.

1. Introduction

Scheduling heuristics find applications in many engineering problems such as silicon compilation (Paulin and Knight 1987), parallel processing (Kasahara and Narita 1984), signal processing (Ashford and Bier 1990), robot dynamics (Chen *et al.* 1988), and many others. This paper presents the solution to the scheduling problem using a genetic approach. As an example, we consider the problem of mapping tasks of a computational algorithm to processing elements. Similar formulations can also be used in other applications. This task assignment plays a very important role in determining the performance of computational algorithms.

Computational algorithms can conveniently be expressed in terms of task graphs (Garey *et al.* 1978). A task graph expresses computation in terms of its constituent computational units, which may be simple arithmetic operations or complex procedures, and the precedence relations among them, namely, $G(\Gamma, \rightarrow, \mu)$, where:

(*a*) $\Gamma(T_1, T_2, \ldots, T_n)$ is the set of tasks (corresponding to nodes of the graph);

(*b*) $\rightarrow$ is the set of edges which represent the precedence relationship between tasks;

(*c*) $\mu(T)$ is the task size, which might be the execution time of the task T on a single processing element p.

In this representation, data transfer times between tasks are assumed to be negligibly short and are therefore ignored. This kind of task graph is referred to as a task graph with no communication costs.

To take advantage of parallelism, it is desired to assign (map) the partially-ordered computational tasks to different processing elements of the system such that the schedule length of the task graph is minimized. This problem is extremely difficult to solve and is generally intractable. It belongs to the class of 'strong' NP-hard problems (Garey *et al.* 1978). Even relaxed or simplified sub-problems constructed by imposing restrictions on the scheduling problem fall into the class of NP-hard problems. The difficulty of obtaining a minimum schedule depends on the number of processing elements, their connectivity, the topology of the task graph, and the uniformity of task processing times.

Received 29 September 1993; accepted 11 March 1994.

†Department of Computer Engineering, King Fahd University of Petroleum and Minerals, Dhahran-31261, Saudi Arabia.

0020–7217/94 $10.00 © 1994 Taylor & Francis Ltd.

The mapping of such task graphs has been studied extensively and several clever heuristics have been proposed (Anger *et al.* 1990, Hironori and Narita 1984, Hwang *et al.* 1989, Kasahara and Narita 1984). Hu (1961) presented a linear time solution for the case where task processing times are all equal and the task graph is tree shaped. Coffman and Graham (1972) proposed an $O(n^2)$ solution to graphs with arbitrary precedence relations, but restricting the number of processing elements to two.

One of the most efficient heuristics for scheduling task graphs with no communication is 'list scheduling'. In this technique a list consisting of ready-to-run tasks is maintained, and no processing element is allowed to remain idle if there is some ready-to-run task that can be executed on it. Whenever a task is added to, or deleted from, the set of ready-to-run tasks, these tasks are sorted in order of priority, so that when a processing element becomes available, it is assigned the task at the head of the list.

Most scheduling heuristics, including list scheduling, are priority based, and they differ only in the way each algorithm assigns priorities to the tasks. The task priority, for example, could be the latest (or earliest) starting time of the task. Some well-known heuristics are the CP (critical path) method (Coffman 1972), HLFET (highest levels first with estimated times), and SCFET (smallest co-level first with estimated times) (Adam *et al.* 1974). An improved version of the CP method of HLFET, called CP/MISF (critical path/most immediate successors first), and an optimization algorithm called DF/IHS (depth first/implicit heuristic search), was proposed by Hironori and Narita (1984).

All of the above methods ignore the overhead of passing data between tasks. This paper addresses the problem of mapping computational tasks to multiple processing elements that would yield a minimal schedule while including the cost of communication between the processing elements. Computational algorithms will be represented by task graphs as $G(\Gamma, \rightarrow, \mu, c)$ where $c(T, T')$ is a measure of the size of the data sent from task T, upon its completion, to its immediate successor task T'. The time to transfer $c(T, T')$ data units depends on the communication media and the inter-connectivity of the processing elements. If the time to transfer a unit of data between two directly-connected processing elements p and q is $r(p, q)$, then, if tasks T and T' are mapped to the processing elements p and q, respectively, then the time to transfer $c(T, T')$ data units is given by $r(p, q) \times c(T, T')$ units. Therefore the execution of T' cannot begin until $r(p, q) \times c(T, T')$ time units have elapsed after execution of T. Note that if $p = q$ then $r(p, q) = 0$; therefore if tasks T and T' are assigned to the same processing element, then the time to transfer data between them is zero and T' can begin execution immediately after the completion of T. Scheduling of T and T' on non-directly connected processing elements will involve a multiplication factor equal to the number of intermediate elements that must relay the results of task T to task T'.

The evaluation of the shortest distance from the entry node to the completion of a task, that is, the earliest completion time (ECT), and from the beginning of a task to the exit node, that is, the latest starting time (LST), is the basis for defining task priorities. This measure indicates which task is more critical than others when the objective function to be minimized is the computation time. Computation of the ECT and LST is straightforward in $G(\Gamma, \rightarrow, \mu)$. However, the evaluation of the ECT and LST are very difficult to obtain for precedence graphs with communication costs, that is, for $G(\Gamma, \rightarrow, \mu, c)$. The reason is that no information is available

beforehand on how paths of the graph are affected by communication requirements. This depends on the mapping of tasks to processing elements and the system interconnectivity. Only those tasks that are mapped to the same processing elements need not account for the communication cost. Therefore, the length of a path is difficult to evaluate before the mapping takes place. The exact values of the ECT and LST depend on determining the optimum finishing time of the computation $G(\Gamma, \rightarrow, \mu, c)$ which is our NP-hard problem (Al-Maasrani 1993).

Prastein (1987) proved that by taking communication into consideration, the problem of scheduling arbitrary precedence graphs on two directly-connected processors is NP-hard. The problem of minimizing the total execution time and communication costs for non-precedence-constrained tasks has been investigated in distributed computing systems (Lo 1984, 1988). Other attempts have been made to find optimal schedules for restricted forms of the $G(\Gamma, \rightarrow, \mu, c)$ model. Anger *et al.* (1990) showed that when there are enough identical processors with identical links, that is $r(p, q)$ is the same for all p and q (except when $p=q$), and when communication delays are no longer than the shortest task processing time, then there is a linear-time optimal algorithm that can solve the scheduling problem. However, this applies only to forests in-trees task graphs on contention-free media. Scheduling heuristics based on the pure use of local priority algorithms for the model with communication tasks have been proposed by Hwang *et al.* (1989). This heuristic is processor driven and uses ETF as the local priority. Some efforts have been made to use the global information. El Rewini and Lewis (1990) used a variation of the 'task level' and included the effect of the task communication as a priority measure. The level of a task is modified to add up all the communication edge values along the longest path from the task node to the exit node. As a result, the level of a task is taken to be the path from the task to any exit node accumulating the highest possible summation of task times and communication edges. El Rewini and Lewis' (1990) results suggest that the use of such measures increases the chance of obtaining better schedules by 1·5 times. Finally, Al-Maasrani (1993) proposed a new global-priority scheduling algorithm based on the evaluation of task finish times in the reverse task graph. Analysis of his results shows the superiority of the heuristic over others reported as shorter finish times are achieved with a stable topology-independent performance.

Optimization methods applied to solve the scheduling problem include branch-and-bound techniques (Kasahara and Narita 1984), and the simulated annealing algorithm (Shield 1987). Other approaches based on clustering techniques have also been reported (Kim and Browne 1988, Sarker and Hennessy 1986). This paper presents a new approach which maps tasks to processing elements based on the genetic algorithm (Holland 1975). This approach, like simulated annealing, is a paradigm for examining a state-space. It produces good solutions through simultaneous consideration and manipulation of a set of possible solutions.

2. Problem statement

Given a set of n computational tasks, the precedence relation among them, m identical processing elements, their inter-connectivity, and the number of data units transferred between tasks, the problem is to determine a schedule with a minimum execution time (scheduling length).

The computation or the set of tasks Γ is represented as a task graph $G(\Gamma, \rightarrow, \mu, c)$. The directed acyclic graph (DAG) G is assumed to have one entry node and one exit

node. It is also assumed that all nodes can be reached from entry and exit nodes. An example of a task graph is given in Fig. 1(*a*). Each node in Fig. 1 is labelled by $T_i/\mu(T_i)$, where T_i represents the task number and $\mu(T_i)$ the task processing time. Arcs directed from node N_i to N_j represent the partial ordering constraint that task T_i precedes task T_j. The weight of the arc between nodes N_i and N_j is $c(T_i, T_j)$, and represents the number of data units passed between tasks T_i and T_j. If the tasks T_i and T_j are scheduled on directly-connected processing elements (say p and q, respectively), then task T_j has to wait for $r(p, q) \times c(T_i, T_j)$ time units before beginning to execute on the processing element q. The value of $r(p, q)$ for any two processing elements p and q in a system depends on its topology and represents the cost of transferring a unit of data from p to q.

Assuming a fully connected system such as the one shown in Fig. 1(*b*), two possible schedules are given in Fig. 1(*c*). In the first schedule, note that task T_5 assigned to processing element p_1 cannot begin execution immediately after T_1 because it has to wait for the completion of task T_3. Since task T_3 is assigned to another processing element, T_5 has to wait for two units of time before beginning execution on p_1. In the second schedule both T_3 and T_5 are assigned to run on the same processing element p_3, therefore T_5 can begin execution immediately after completion of T_3.

We assume contention-free media, that is, the communications channel between processing elements is assumed to have sufficient capacity to serve all transmissions without delay. Therefore, issues related to routing and queuing are ignored. We also assume that processing elements are identical, although a similar technique can be extended to non-identical ones.

2.1. *Problem formulation*

This paper formulates the problem of mapping tasks to processing elements as a combinatorial problem. There are m^n possible assignments of m or less processing

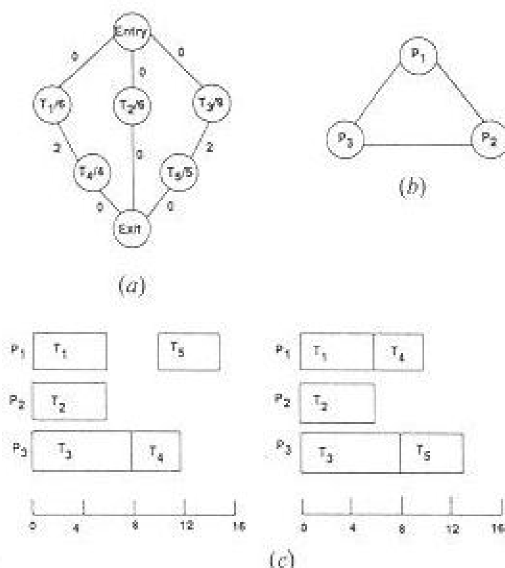

Figure 1. (*a*) Task graph; (*b*) topology of processing elements; (*c*) possible schedules.

elements to execute n tasks of a task graph. The search space for determining the optimal assignment is very large, which rules out any greedy or brute-force technique, and as the number of tasks increases the search space of possible assignments grows exponentially. In the next section, a genetic algorithm will be used to solve this combinatorial optimization problem. This approach, like simulated annealing, is a paradigm for examining a state-space. It produces good solutions through simultaneous consideration and manipulation of a set of possible solutions.

3. Genetic scheduling algorithm

Genetic algorithms (GAs) are powerful domain-independent search mechanisms which emulate the natural process of evolution as a means of progressing towards the optimum (Holland 1975). They have been applied in solving various optimization problems, including those in VLSI physical design (Cohoon and Paris 1987, Shahookar and Mazumder 1990).

In the GA approach, at any given instance a number of possible solutions, called the 'population', exist. Their number, denoted by N_p, depends on the problem instance, the size of the problem, and the available memory. Each individual in the population is a string of symbols. These symbols are known as 'alleles' and the string made up of these alleles is termed 'chromosome'. The chromosome represents a possible solution to the optimization problem. During each iteration (generation), the individuals in the current population are evaluated using some measure of 'fitness'. Based on the fitness value, two individuals at a time (called 'parents') are selected (ϕ) from the population. The more 'fit' the individual the higher the probability of it being selected. Then genetic operations are applied on the selected parents to generate new individual solutions called 'offsprings'. These genetic operators combine the features of both parents. Common operators are crossover and mutation. They are derived by analogy from the biological process of evolution. To apply genetic algorithms as an adaptive search strategy, the problem must be mapped into a representation suitable for genetic search. First we shall see how an assignment (schedule) is mapped to a string. Then, we will discuss the basic operators applied in search for an optimum schedule. The structure of the genetic algorithm employed is given in Fig. 2.

3.1. *String encoding* ξ

One of the most important steps in genetic algorithms is to represent the solution as a string of symbols. This encoding must be amenable to genetic operations.

In our problem, each solution or schedule can be encoded as a string containing n alleles, where n is the number of tasks.The encoding divides each allele into two fields T_j and p_k; T_j, $1 \leq j \leq n$, and p_k, $1 \leq k \leq m$. Field p_k specifies the processing element number and field T_j specifies the task that is assigned to it. For example, for our first schedule in Fig. 1(c) we have the encoding $[(T_1, p_1), (T_2, p_2), (T_3, p_3), (T_4, p_3), (T_5, p_1)]$.

Since the permutation of these alleles also represents the same schedule, it is convenient to keep the elements of the string sorted on the order of task indices. This makes the string amenable to genetic operators. If we keep the alleles sorted on the task indices, since it is not required to store the indices, the string $[p_1, p_2, p_3, p_3, p_1]$ is sufficient to represent our schedule.

```
Procedure (Genetic_Algorithm_For_Scheduling)
    Np= Population size.                (*# Of possible solutions at any instance.*)
    Ng= Number of generations.          (*# Of iterations.*)
    No= Number of offsprings.           (*To be generated by crossover.*)
    Nm= Number of mutated parents.
    Pμ= Mutation probability.
    P ← Ξ(Np)                           (*Construct initial population P.*)
    For j = 1 to Np
            Evaluate σ(P[j])            (*Evaluate fitness of P which is*)
                                        (*the reciprocal of length of schedule.*)
    EndFor
    For i = 1 to Ng
            For j = 1 to No
                (x, y) ← φ(P)           (*Probability of parent selected is proportional to its fitness.*)
                offspring[j] ← ψ(x, y)  (*Generate offspring by simple crossover.*)
                Evaluate σ(offspring[j])
            EndFor

            For j = 1 to Nm             (*With probability Pμ apply mutation.*)
                mutated[j] ← μ(y)
                Evaluate σ(mutated[j])
            EndFor
    P ← Select(P, offspring, mutated)   (*Select best Np solutions from parents, offsprings*)
                                        (*and mutated parents.*)
    EndFor
    Return highest scoring configuration in P.
End
```

Figure 2. Genetic algorithm for scheduling.

3.2. *Population constructor* Ξ

The quality of final solution depends upon the size of the population and how the initial population is constructed. One possibility is to ignore the communication cost and use a priority-based constructive heuristic to find the initial assignment. Another possibility is to begin with a random assignment. The technique adopted to construct the initial population randomly is explained below.

To construct an initial set of solutions for a task graph with n tasks, N_p random permutations of numbers from 1 to n are generated. The number in each random permutation represents the task number of the task graph. Then tasks are assigned to processing elements uniformly, the first $\frac{n}{m}$ tasks in our permutation are assigned to processing element p_1, the next $\frac{n}{m}$ tasks to processing element p_2, and so on. This is referred to in the literature as pre-scheduling (Benten 1989). For example, let the given graph contain nine nodes and let there be three processing elements. If one of the random permutations is [7, 1, 8, 3, 4, 9, 5, 6, 2], then the assignment of processing elements will be [(7, p_1) (1, p_1) (8, p_1) (3, p_2) (4, p_2) (9, p_2) (5, p_3) (6, p_3) (2, p_3)], that is, the first three tasks are assigned to p_1, the next three to p_2 and the last three to p_3. Sorting our assignment on tasks, the equivalent assignment is [(1, p_1) (2, p_3) (3, p_2) (4, p_2) (5, p_3) (6, p_3) (7, p_1) (8, p_1) (9, p_2)]. Therefore, our chromosome representation, or the solution string, is [p_1, p_3, p_2, p_2, p_3, p_3, p_1, p_1, p_2]. Note that when the algorithm performs crossover the distribution of tasks to processing elements will no longer be uniform.

3.3. *Crossover operator* ψ

Crossover is the main genetic operator. It operates on two parents, one called the 'target' parent and the other called the 'passing' parent, and generates an offspring.

Crossover is an inheritance mechanism where the offspring inherits the characteristics of the parents.

A simple crossover operation performs the 'cut-paste-and-patch' operation. It consists of choosing a random cut point and generating the offspring by combining the segment of one parent (string) to the left of the cut point with the segment of the other parent to the right of the cut point. Our string encoding function ξ is such that this simple technique produces valid solutions or legal schedules.

As an example, let the passing parent be $[p_3, p_1, p_2, p_1, p_3, p_2, | p_1, p_3, p_1]$ and the target parent be $[p_2, p_3, p_1, p_2, p_2, p_2, | p_2, p_2, p_3]$. If the crossover point is chosen after the fifth allele, then the offspring produced will contain the alleles from the left of the crossover point of the passing parent and those from the right of the target parent. Our new assignment of tasks to processing elements will thus be represented by the offspring produced, given by $[p_3, p_1, p_2, p_1, p_3, p_2, p_2, p_2, p_3]$.

Several crossover operators were attempted, but the simple crossover technique explained above was found to be genetically effective and computationally cheap. This simple crossover technique tends to preserve some of the characteristics of the parent chromosomes, and always produces valid schedules. Note that without loss of generality, tasks can be re-numbered in such a way that the index of the parent task is always less than the index of its children. Tasks in the same level are also sorted, with the left-most one taking the lowest index. In this case the above simple crossover operator will yield an offspring schedule that uses the target parent's assignment of the upper part of the task graph combined with the passing parent's assignment of the lower part of the graph.

3.4. *Mutation function μ*

Mutation produces incremental random changes in the offspring generated by the crossover. In our case, a simple mutation mechanism is the pair-wise swap of assignment of tasks to processing elements. That is, if task T_i was assigned to processing element p_l and task T_j to processing element p_k, then after mutation task T_i will be assigned to p_k and task T_j to processing element p_l.

Mutation is not applied to all members of our population, but is applied probabilistically to some members. For a given assignment, two tasks are randomly chosen and their assignments are swapped as explained above.

Mutation is important because crossover alone will not help to obtain a good solution. Crossover is only an inheritance mechanism. The mutation operator generates new characteristics. If the new offsprings perform well, then the configurations containing them are retained and these spread throughout the population. The mutation rate controls the rate at which new genes are introduced into the population for trial. If the mutation rate is low then many genes that would have been good are never tried out. If the mutation rate is high then there will be too much random disturbance, causing the offsprings to lose resemblance to their parents, and the algorithm will lose its ability to learn from the history of the search.

3.5. *Fitness function σ*

The fitness function, also known as the scoring function of a solution, is the objective function of this combinatorial optimization problem that we want to maximize. It is obviously the reciprocal of the elapsed execution time of the task graph using a given processing element task assignment. Hence, the assignment that

has the smallest elapsed execution time represents the most fit individual of the population. For a given assignment of tasks to processing elements, this value is expressed as the maximum of the sum of the time required to execute all the tasks assigned to a given processing element, plus the idle time of that element.

$$\sigma^{-1} = \max_{j=0,m-1} \left(\sum_{T_i \in p_j} T_i + \text{idle}(j) \right) \tag{1}$$

To compute the time to completion the following procedure is used. First, the given graph is levelled in such a way that no task and its parent (or child) appear in the same level. This can be accomplished by the insertion of dummy nodes between tasks that will make the level of the child of a node one more than its parent. In this case the number of nodes from entry to exit along all paths will be the same. Dummy nodes are assigned to dummy processing elements and their tasks take zero time to execute. Then at each level, the time to complete the tasks at that level for the partial schedule is determined. In case more than one node in a given level is assigned to the same processing element, the order of execution of these tasks is such that sizes of gaps between tasks is reduced. Note that for such nodes there are no precedence constraints since they are in the same level of the levelled graph. It must be mentioned that dummy nodes are assigned task numbers which are greater than the maximum task index in the original task graph. Since crossover and mutation are applied to only the first n alleles of the chromosome, addition of dummy nodes does not increase the search space for the best schedule.

4. Results and discussion

4.1. *Implementation issues*

The genetic algorithm was coded in the C language with a little over 600 lines of code. The core of the genetic algorithm does not exceed 200 lines and consists of a procedure to level the task graph, a short crossover routine and a similar mutation procedure. There is also a function that calculates the fitness function σ. The program reads in a task graph represented by a data set that consists of numbered tasks and their associated costs, parents, and the cost of communication with these parents. The data also designates the start node and the exit node of the task graph. The program then begins by levelling the task graph and adding any required dummy nodes as described earlier, and generates an initial set of size N_p processing elements assigned to the tasks. The number of processing elements in a run-time input parameter that is supplied by the user. For the random graphs generated for our experiment the value of $N_p = n$ was a reasonable choice for $10 < n < 40$ and $3 < m < 7$.

The genetic algorithm begins by computing the fitness of the initial set of assignments which will be referred to as the parent set. Using this population, a set of offspring processing element assignments are obtained by crossing pairs of the parent set N_p times. Another set of offsprings of size N_p is also computed by mutating the parent set. Upon completion of computing the fitness of the newly generated sets, the three sets are sorted and the best N_p values form the new population of the next generation. This process is repeated until a user-specified condition is satisfied. In our experiment the program terminates if there is no improvement in the average fitness of the population for the last $10 \times n$ generations or when the optimal value is reached (see Fig. 9).

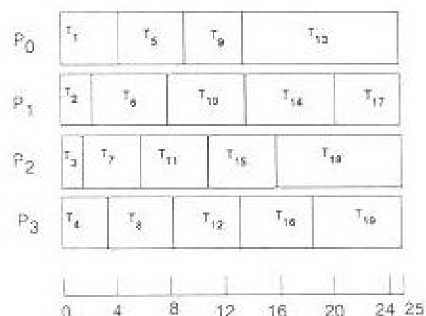

Figure 3. Random Gantt chart.

The program has been run on several platforms that include a MIPS Decstation, Sun 10 and an Alpha Decstation with numerous examples.

For task graphs intended for execution on dedicated VLSI systems, or parallel processors such as the Hypercube or a CM* machine, preliminary experiments have shown that the order of execution times were reasonably good, given the size of the search space. Finally, the implementation includes a subroutine tht produces the Gantt chart as well as the task graph which are output in PIC format.

4.2. *Solution quality*

It is natural to ask how good is the solution generated by the genetic heuristic? Assume that S_A is the solution generated by our heuristic algorithm developed for the minimization problem. If S^* is the optimum solution, a measure of the error (ε) made by the heuristic is the relative deviation of the heuristic solution from the optimal solution, that is

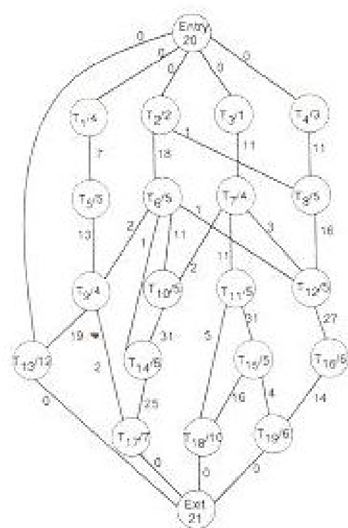

Figure 4. Random task graph corresponding to Gantt chart of Fig. 3.

```
 Task graph file = bench2    No of tasks = 28     No of procs = 4
Solution={3 2 0 3 3 0 1 2 3 0 2 2 1 0 2 1 2 0 1 | 1 3 1 2 3 1 0 0 3 } time = 74
Solution={0 2 3 2 3 0 2 3 3 3 1 3 1 3 1 2 3 3 2 | 2 2 0 0 1 1 0 1 3 } time = 70
Solution={2 3 1 0 0 3 1 2 0 2 1 2 1 1 0 2 1 0 2 | 0 0 3 0 1 0 3 1 0 } time = 68
Solution={0 2 3 3 3 0 2 3 3 3 1 2 1 3 1 2 3 3 2 | 2 2 0 0 1 0 3 3 3 } time = 64
Solution={2 3 1 0 0 3 1 2 0 0 2 2 1 0 2 2 1 0 2 | 0 0 1 2 3 1 0 3 3 } time = 63
Solution={1 0 2 2 3 0 2 3 3 3 1 2 1 3 1 2 3 3 2 | 2 2 0 0 1 1 0 1 3 } time = 56
Solution={1 0 2 2 3 0 2 2 3 3 1 2 1 3 1 2 3 3 2 | 2 3 0 0 3 2 3 3 1 } time = 55
Solution={1 0 2 0 3 0 2 2 3 3 1 2 1 3 1 2 3 3 2 | 0 1 2 3 1 1 0 2 3 } time = 52
Solution={3 0 2 3 3 0 2 3 3 3 1 2 3 3 1 2 3 1 2 | 0 2 3 0 1 0 3 1 3 } time = 51
Solution={3 0 2 3 3 0 0 2 3 3 1 2 0 3 1 2 3 1 2 | 0 2 0 1 1 1 3 3 0 } time = 47
Solution={3 0 0 3 3 0 0 2 0 3 1 2 0 3 1 2 3 1 2 | 1 0 2 1 1 0 3 3 3 } time = 45
Solution={3 0 0 3 3 0 1 2 0 3 1 2 0 3 1 2 3 1 2 | 1 0 2 1 1 0 0 3 0 } time = 41
Solution={3 0 1 1 3 0 1 2 0 3 1 2 0 3 1 2 3 1 2 | 1 0 2 3 3 3 1 1 0 } time = 38
Solution={3 0 1 1 0 0 1 2 0 3 1 2 0 3 1 2 3 1 2 | 3 1 1 1 3 3 1 1 0 } time = 37
Solution={3 0 2 1 0 0 1 2 0 3 1 2 0 3 1 2 3 1 2 | 1 1 0 0 3 3 3 1 1 } time = 36
Solution={3 3 1 2 0 3 1 2 0 3 1 2 0 3 1 2 3 1 2 | 3 1 0 0 0 3 3 1 1 } time = 32
Solution={0 3 1 2 0 3 1 2 0 3 1 2 0 3 1 2 3 1 2 | 3 1 0 3 1 3 3 1 1 } time = 25
```

Figure 5. Solution string after each iteration of the most fit individual generated by the genetic algorithm for the random task graph of Fig. 4.

$$\varepsilon = \frac{S_A - S^*}{S^*} \tag{2}$$

Unfortunately, it is not easy to measure the error, since S^* is not known. The problem of computing S^*, as seen above, is as hard as finding the optimum solution itself! Therefore, we have to resort to other techniques for judging the quality of solutions generated by our heuristic algorithms.

One method to alleviate the above problem is to artificially generate test inputs for which the optimum solution is known *a priori*. We generated random schedules whose minimal finish time is known, and from these schedules we constructed the

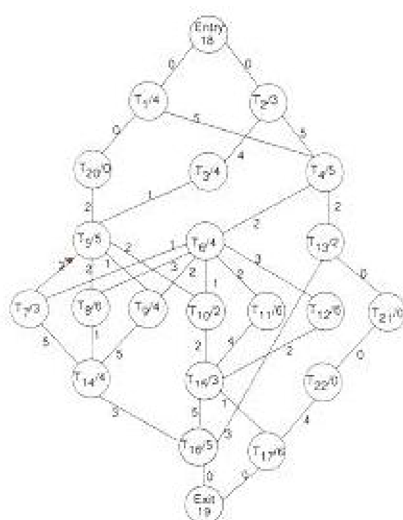

Figure 6. Levelled task graph from Al-Mouhamed (1990).

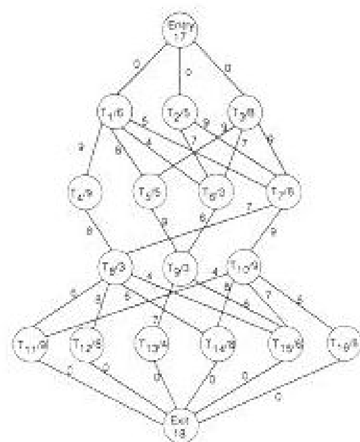

Figure 7. Task graph from Al-Maasrani (1993).

required task graphs. The procedure to generate random tasks for which the optimal finish time is known *a priori* is as follows.

We assume that the number of processing elements and the time to completion is given. The time interval between zero and finish time is divided randomly into slices, that is, we have generated a random Gantt chart. Each slice corresponds to a task and its width represents the time to completion of that task on the given processing element. To generate random task graphs we have to generate nodes and edges. Each slice corresponds to a node (task) in the task graph. Next, edges are added between tasks (nodes) as follows. Several pairs of tasks (say T_i and T_j) are chosen randomly, and if the finish time of task T_i in our Gantt chart is before the starting time of task T_j, an arc is added between them in the task graph. A communication cost is assigned to this arc, whose value is equal to or less than the separation between the finish time of task T_i and start time of T_j. If T_i and T_j are on the same processing element then any arbitrary cost is assigned to this edge.

Now that we know the best solution for our task graph, we use this task graph as input to our genetic heuristic.

An example Gantt chart is shown in Fig. 3. The two tasks T_2 and T_6 are selected, and since they are assigned to the same processing element, an edge with an arbitrary cost (18) is added between them. For tasks T_7 and T_{10} the difference in the finish time of T_7 and start time of T_{10} is 2 units, so an edge with a cost of 2 units or less may be added between these nodes. The generated random graph corresponding to the random Gantt chart is given in Fig. 4.

The solution string of each iteration of the most fit individual generated by the genetic algorithm for the random task graph of Fig. 4 is given in Fig. 5. Note that the first 19 alleles correspond to 19 tasks of the task graph. The remaining nine alleles correspond to the dummy nodes added to level the graph to ease the computation of *time to completion*. Observe that the assignment produced by our genetic procedure is different from the original one, although the time to completion obtained is the same.

We also used as examples some task graphs available in the literature. For the task graph given by Al-Mouhamed (1990) (see Fig. 6) the predicted lower bound on

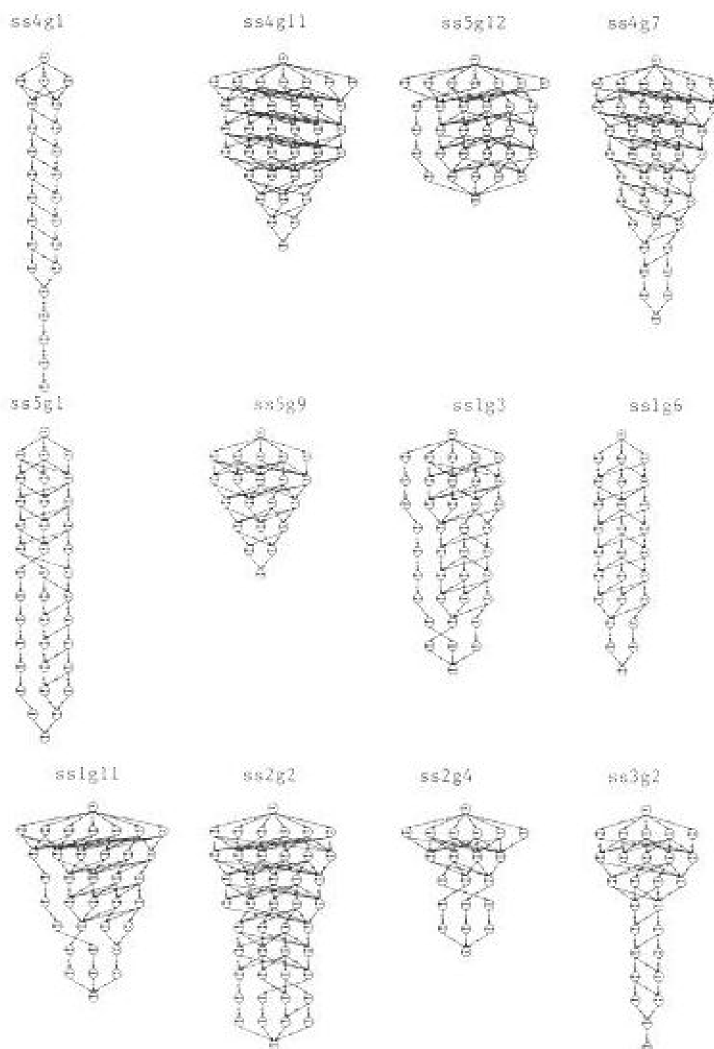

Figure 8. Random sample task graphs.

finish time was 35 with seven processors. Our heuristic produced a time-to-completion of 38 with seven processors and 39 with only three processors. The three-processor assignment solution expressed as our encoded string is $[p_0, p_2, p_1, p_0, p_1, p_0, p_0, p_1, p_1, p_2, p_2, p_0, p_1, p_0, p_2, p_0, p_1]$. For another example taken from Al-Maasrani (1993) (see Fig. 7) our heuristic provided a finish time of 46 units, better than the finish time obtained by other constructive heuristics which are as follows: PD/ETF (53 units), LST (50 units), GD/HLETF (50 units), and GD/HLETF* (47 units). The assignment of this solution expressed as our encoded string is $[p_2, p_0, p_1, p_1, p_0, p_0, p_2, p_1, p_2, p_0, p_1, p_0, p_2, p_0, p_1, p_2, p_1]$.

For all the task graphs, our results were equal to or better than those produced by other approaches. A sample of randomly generated task graphs whose optimal time to completion is known are shown in Fig. 8. The Table compares the solution

obtained using the genetic heuristic with the optimal. It also lists solutions obtained using a straight ETF scheduling algorithm. The quality of solution in all cases is well within 80% of the optimal, and the average quality is within 90% of the optimal. The time to execute the genetic procedure on an Alpha Decsystem 3000/400 is of the order of seconds. The Table also shows the number of generations required, and the

Graph	m	n	T_{opt}	T_{gas}	t_{exec}	N_g	Solution$_Q$	ETF	One processor
maz1	19	7	35	38	3	366	0·92	48	72
ss1g2	30	5	84	97	21	456	0·87	150	246
ss1g6	26	3	84	84	2	128	1·00	147	206
ss1g11	28	7	63	63	17	502	1·00	103	262
ss2g2	40	6	91	109	57	861	0·83	157	335
ss2g3	36	5	91	109	26	503	0·83	141	290
ss2g4	17	6	40	41	3	317	0·98	72	122
ss3g2	27	6	106	113	15	781	0·94	186	248
ss3g11	26	7	52	65	6	334	0·80	103	214
ss4g1	25	3	129	133	5	328	0·97	245	209
ss4g7	40	6	99	117	48	848	0·85	197	379
ss4g11	36	7	82	94	44	1080	0·87	149	324
ss5g1	30	3	128	136	20	483	0·94	241	276
ss5g7	39	6	103	119	43	810	0·87	207	375
ss5g9	21	5	66	68	4	320	0·97	115	181
ss5g12	28	7	66	74	15	543	0·89	120	258

Comparing the results of the genetic algorithm with optimal values.

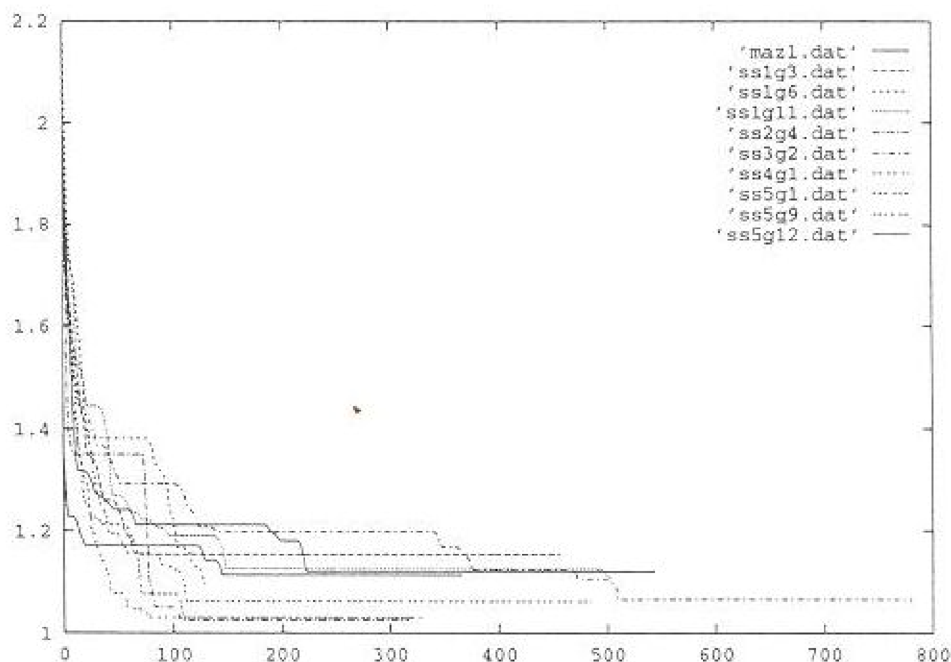

Figure 9. Plot of number of generations versus (Average $T_{finish}/T_{optimal}$).

single processor execution time to demonstrate the possible gains and speed-up that can be achieved. Note that for the task graph ss4g1 the ETF algorithm may slow down the execution of this task graph on a multiprocessor if its solution is used. The Table also shows several cases where the optimal, or very near optimal, scheduling assignment is achieved.

Figure 9 shows the improvement in the average fitness of the population from one generation to the next. This graph clearly shows that our genetic heuristic converges towards an optimal solution.

5. Conclusions

This paper has applied a genetic algorithm to find a minimal schedule assignment of tasks to processing elements on multiprocessing systems. The algorithm is topology-independent and can be applied to schedule tasks on dedicated VLSI circuits, Hypercubes, Grid-connected or randomly-connected multiprocessors. This technique can also be applied to schedule task graphs with no communication costs, in job-shop scheduling problems, load balancing in operating systems for multiprocessors, and in optimization of other combinatorial problems. Experiments on large, randomly generated task graphs show that the procedure is very effective in determining the minimal time to completion. In all cases tested, results obtained are comparable or better than earlier approaches.

Acknowledgments

The authors acknowledge King Fahd University of Petroleum and Minerals for all support. We would like to thank the staff of KFUPM library for supplying the requested literature.

References

Adam, T. L., Chandy, K. M., and Dickson, J. R., 1974, A comparison of list schedules for parallel processing systems. *Communications of the Association of Computing Machinery*, **17**, 685–690.

Al-Maasrani, A. M. A., 1993, Priority-based scheduling and evaluation of precedence graphs with communication times. M.S. thesis, King Fahd University of Petroleum and Minerals, Saudi Arabia.

Al-Mouhamed, M. A., 1990, Lower bounds on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Transactions on Software Engineering*, **16**, 1390–1401.

Anger, F. D., Hwang, J. J., and Chow, Y. C., 1990, Scheduling with sufficient loosely coupled processors. *Journal of Parallel and Distributed Computing*, **9**, 87–92.

Ashford, L., and Bier, J., 1990, Architectures for statistically scheduled dataflow. *Journal of Parallel and Distributed Computing*, **10**, 333–348.

Benten, M. S. T., 1989, Multiprogramming and the performance of parallel programs. Ph.D. dissertation, University of Colorado, Boulder, U.S.A.

Chen, C. L., Lee, C. S., and Hou, E. S. H., 1988, Efficient scheduling algorithm for robot inverse dynamics computation on a multiprocessor system. *IEEE Transactions on Systems, Man and Cybernetics*, **18**, 729–742.

Coffman, E. G., 1972, *Computer and Job-Shop Scheduling Theory* (New York: Wiley).

Coffman, E. G., and Graham, R. L., 1972, Optimal scheduling for two processor systems. *Acta Informatica*, **1**, 200–213.

Cohoon, J. P., and Paris, W. D., 1987, *Genetic placement. IEEE Transactions on Computer-Aided Design*, **6**, 956–964.

Garey, M. R., Graham, R. L., and Johnson, D. S., 1978, Performance guarantees for scheduling algorithms. *Operations Research*, **26**, 3–21.

Hironori, K., and Narita, S., 1984, Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, 1023–1029.

Holland, J. H., 1975, *Adaptation in Natural and Artificial Systems* (Ann Arbor, Mich., U.S.A.: University of Michigan Press).

Hu, T. C., 1961, Parallel sequencing and assembly line problem. *Operations Research*, **9**, 841–848.

Hwang, J. J., Chow, Y. C., Anger, F. D., and Lee, C. Y., 1989, Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 244–257.

Kasahara, H., and Narita, S., 1984, Practical multiprocessor scheduling algorithm for efficient parallel processing. *IEEE Transactions on Computers*, **33**, 1023–1029.

Kim, S. J., and Browne, J. C., 1988, A general approach to mapping of parallel computation upon multiprocessor architecture. *Proceedings of the International Conference on Parallel Processing*, **3**, 1–8.

Lo, V. M., 1984, Heuristic algorithm for task assignment in distributed systems. *Proceedings of the 4th International Conference on Distributed Computing Systems*, pp. 3–21; 1988, Heuristic algorithm for task assignment in distributed systems. *IEEE Transactions on Computers*, **37**, 1384–1397.

Paulin, P. G., and Knight, J. P., 1987, Force-directed scheduling in automated data path synthesis. *24th Design Automation Conference*, pp. 195–202.

Prastein, M., 1987, Precedence-constrained scheduling with minimum time and communication. M.S. thesis, University of Illinois at Urbana-Champaign, U.S.A.

El Rewini, H., and Lewis, T. G., 1990, Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, **9**, 138–153.

Sarker, V., and Hennessy, J., 1986, Compile-time partitioning and scheduling of parallel programs. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pp. 17–26.

Shahookar, K., and Mazumder, P., 1990, A genetic approach to standard cell placement using meta-genetic parameter optimization. *IEEE Transactions on Computer-Aided Design*, **9**, 500–511.

Shield, J., 1987, Partitioning concurrent VLSI simulated programs onto multiprocessor by simulated annealing. *Proceedings of the Institute of Electrical and Electronics Engineers*, **134**, 24–30.