A Genetic Algorithm for Scheduling Directed Acyclic graphs in The Presence of Communication Contention

Yoginder S. Dandass[a]

[a]Mississippi State University, Department of Computer Science, PO Box 9637, Mississippi State, MS 39762
yogi@hpcl.cs.msstate.edu

This paper presents an algorithm for scheduling parallel applications represented in the form of directed acyclic graphs (DAGs) onto a set of homogeneous parallel processors. The algorithm utilizes a genetic algorithm to prioritize scheduling activities for a list scheduling algorithm with the primary goal of minimizing schedule length and a secondary goal of minimizing the total number of processors used in the schedule. The genetic list scheduling (GLS) algorithm presented in this paper extends existing research by considering contention for processor-to-network links during communication operations. This requires the GLS to efficiently schedule both communication and computation operations in order to reduce schedule lengths. Furthermore, The GLS presented in this paper also strives to efficiently schedule multicast operations over a virtual point-to-point topology. Schedules produced by this GLS algorithm are shown to have shorter lengths than those produced by the HLEFT and ETF algorithms that have been modified to handle communication contention and multicast communication. A parallel implementation of the GLS algorithm using the synchronous connected island model is also investigated and is shown to produce better results than the sequential implementation.

Cet article présente un algorithme d'ordonnancement d'applications parallèles, représentées par des graphes à orientation acyclique (directedacyclic graphs ou DAGs), sur un ensemble homogène de processeurs parallèles. L'approche utilise un algorithme génétique qui priorise l'ordonnancement de façon à réduire le délai d'attente et minimiser le nombre de processeurs utilisés. L'approche présentée (genetic list scheduling ou GLS) permet d'augmenter l'étendu de nos connaissances actuelles avec la prise en compte de l'embouteillage du lien processeurs-réseau pendant les opérations de communication. L'algorithme GLS doit donc ordonnancer efficacement à la fois la communication et les calculs. De plus, la variété de GLS présentée dans cet article prend en compte le plus efficacement possible les opérations de multidiffusion sur topologie réseau point à point. Cet algorithme GLS donne lieu à de plus court délais que des algorithmes HLEFT et ETF ayant été modifiés pour prendre en considération les embouteillages réseau et les multidiffusions. Une implémentation parallèle de l'algorithme GLS (synchronous connected island model) a également été expérimentée et a produite de meilleurs résultats que l'implémentation séquentielle.

1 Introduction

Parallel processing is becoming a popular approach for solving large problems whose processing requirements exceed the computational and memory capacity of individual processors. In order to make economical use of system resources, parallel applications require efficient scheduling of their computation and communication tasks in resource-constrained systems (*e.g.*, systems with limited number of processors and network bandwidth). Therefore, fine-grained scheduling and resource allocation of parallel application with the goal of increasing system efficiency has become an important area of research.

The computational requirements of the constituent tasks for many parallel applications can be estimated using analytical and empirical techniques, and the analysis of the control and data flow of these applications can provide information regarding inter-task precedence relations and communication requirements [6, 8]. This information, represented in the form of direct acyclic graphs (DAGs), can be used to construct efficient static schedules for these applications prior to their execution in production environments where minimizing runtimes is critical.

Scheduling DAGs with the goal of minimizing *makespan* (*i.e.*, schedule lengths) is known to be NP-hard in general [5]. Therefore, a number of heuristic approaches have been proposed that can produce near-optimal schedules in polynomial time [2, 9]. List scheduling (LS) is one such heuristic technique that has been extensively studied, and a large number of LS heuristics have been proposed [1, 15]. Genetic algorithms (GAs) have also been applied to solve scheduling problems and hybrid algorithms combining list scheduling with GA (called genetic list scheduling) have recently been developed [11, 12, 16].

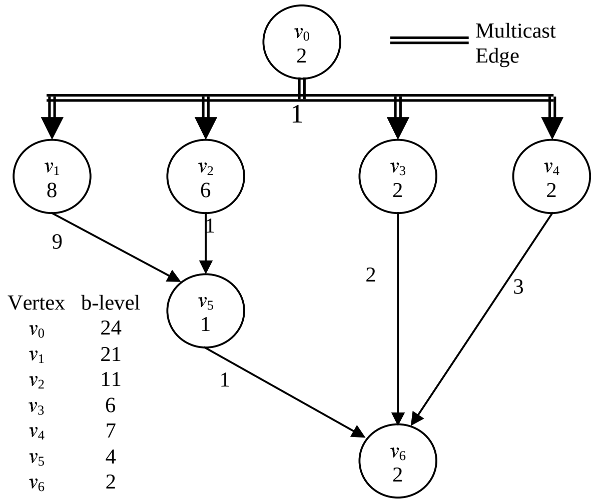
Most LS research efforts make the simplifying assumption of having completely contention-free communication operations (*i.e.*, a point-to-point physical network is assumed). This assumption allows the list schedulers to focus on scheduling computational tasks without considering the interaction of communication operations competing for limited network resources. While a high-capacity network fabric (*i.e.*, switches and switch interconnects) may appear to be contention free, contention over processor-to-switch links cannot be disregarded when scheduling communication operations. This is because communication delays due to link contention can, in turn, delay the execution of dependent computational tasks.

This paper presents a genetic list scheduling (GLS) algorithm, that takes communication contention into account when constructing schedules from DAGs. In the face of competing computation and communication tasks, the algorithm proposed in this paper strives to optimize the scheduling of both computation and communication.

The contention free communication assumption in prior research enabled all the edges originating from and arriving at a vertex to be scheduled to occur simultaneously. This obviated the need for special handling of multicast operations. In a multicast operation, identical data appears to be transmitted from the *root* (*i.e.*, source) vertex to all recipient vertices. The GLS algorithm presented here parallelizes the logical point-to-point operations underlying the multicast operation by considering all processors that have copies of the data to be potential sources. This parallelization can significantly reduce schedule makespan as compared to schedules with serialized point-to-point multicast implementations. The algorithm presented in this paper uses an extended DAG representation in order to enable the specification of multicast edges.

The remainder of this paper is organized as follows: Section 2 provides a detailed problem definition, describes the parallel processing environment, and identifies the simplifying assumption made regarding the communication infrastructure. Section 3 provides a brief overview of LS and GLS techniques. Section 4 describes the GLS approach used in this research. Section 5 presents experimental results and analysis. Section 6 highlights key contributions and discusses of avenues for further research.

2 Problem Definition



Vertex	b-level
v_0	24
v_1	21
v_2	11
v_3	6
v_4	7
v_5	4
v_6	2

Figure 1: Hypothetical DAG and vertex b-levels

A DAG $G = \{V, E\}$ consists of a set $V = \{v_1, v_2, \ldots, v_n\}$ of n vertices and a set $E = \{e_1, e_2, \ldots, e_k\}$ of k directed edges connecting the vertices. The vertices represent computational tasks in a parallel application and the edges represent communication and precedence relations between the tasks. The ordered pair $e_i = (v_{src}, v_{dest})$ indicates that the direction of edge e_i is from vertex v_{src} to v_{dest}. In Figure 1, depicting a simple DAG, edges (v_0, v_1), (v_0, v_2), (v_0, v_3), and (v_0, v_4) are multicast edges.

The primary objective of the algorithm described in this paper is to schedule DAGs onto a parallel machine so as to minimize its *makespan* (*i.e.*, schedule length) while utilizing a “reasonable” number of processors. Of two schedules with identical makespans, the schedule requiring fewer processors is preferred.

The processor time $t(v_i)$ required to complete the computation of node v_i is given by its weight $w(v_i)$. For example, in Figure 1, $t(v_0) = w(v_0) = 2$. Because communication cost between vertices scheduled on the same processor is assumed to be negligible, the time to execute e_i is given by

$$t(e_i, p_s, p_d) = \begin{cases} 0 \text{ when } p_s = p_d \\ w(e_i) \text{ otherwise} \end{cases},$$

where p_s, and p_d are the processors on which the source and destination vertices of e_i are scheduled, respectively.

A homogeneous parallel environment consisting of identical processors is assumed for executing the parallel application. Therefore, the time taken to execute a computational task is the same on any processor. Also, a uniform network capacity is assumed over the entire parallel system. This implies that the time needed to complete a particular point-to-point communication operation is the same over any combination of source and destination processors.

Each processor is assumed to have a half duplex interface to the homogeneous virtual point-to-point network. This restricts communication to either one incoming operation or one outgoing communication operation at each processor-network link at a time. Conversely, the switched network fabric is assumed to be contention free (*i.e.*, there is sufficient network capacity available to ensure that the various communication operations do not interfere with each other). This is a reasonable assumption because high-performance switches can multiplex several messages simultaneously, and alternative routes can be used to avoid congestion.

This research also considers efficient scheduling for multicast operations over a virtual point-to-point network. Instead of using a single spanning tree-based pattern of point-to-point operations for all instances of multicast communication as proposed in [4], a GLS is used to discover the best possible schedule for the best possible set of point-to-point operations that can be used to implement each multicast edge.

This research assumes that computation and communication may be overlapped without affecting each other’s completion time. In practical systems, contention over the shared system and memory busses between the processor the network interface card (NIC) during DMA operations can cause variances in communication and computation completion times. However, these variances are assumed to be sufficiently small so as to have minimal impact on the schedule.

It is also assumed that communication operations need not be tightly coupled between sending and receiving tasks (*i.e.*, the sending and receiving computation tasks need not be executing simultaneously with each other or during the associated communication operations). Given a schedule, the system initiates data transfer at the appropriate time and buffers data that cannot be immediately transmitted or consumed. Note that precedence is still observed; the destination task cannot begin execution before all incoming communication operations are completed. The overhead of buffer manipulation can be reduced through the use of efficient buffer management semantics (*e.g.*, similar to the buffer management facilities provided by MPI/RT [18]).

3 Background

3.1 List Scheduling

The fundamental LS process is the following:

1. Construct a *ready list* of vertices with no preceding vertices.
2. While vertices remain in the ready list:
   3. Prioritize the ready list.
   4. Remove the highest priority vertex from the ready list and schedule it on the processor that will allow the earliest start time for this vertex.
   5. Add the newly readied vertices to the ready list.

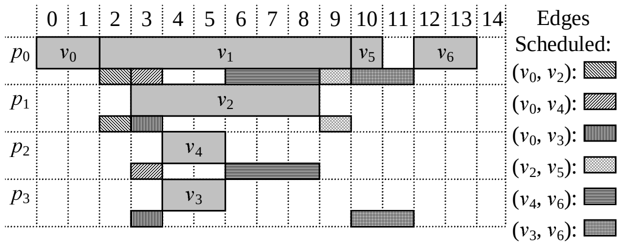
A ready vertex is one whose precedence constraints have been met. In typical LS algorithms, vertex priorities are determined either statically from vertex attributes before the scheduling process begins or dynamically during the scheduling process. The variety of attributes and ways in which they are applied in various LS approaches are too numerous to mention here. The survey of LS algorithms by Kwok and Ahmad [15] provides such details.

Of all LS algorithms, the *Dynamic Critical Path* (DCP) algorithm [17] appears to have the best performance over a large number of sample DAGs. However, as with most other LS algorithms, DCP assumes contention free communication and does not consider multicast operations. Unfortunately, the complex nature of DCP precludes its straightforward modification to address these issues. Therefore, in order to provide a basis for evaluating the GLS approach proposed here, two simpler LS algorithms were modified to handle communication contention and multicast edges.

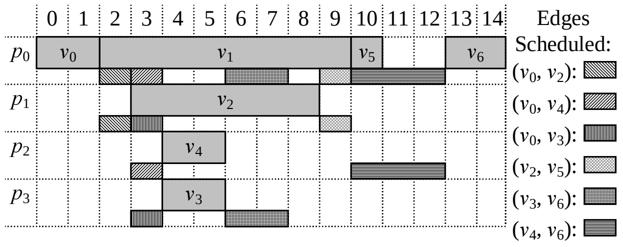
The *Highest Level First with Estimated Times* (HLEFT) [1] algorithm is a simple (and consequently a fast) LS algorithm. It prioritizes vertices in the ready list in descending order of their *static b-level* attributes. The *static b-level* of a node n_i is the length of the longest path between n_i and an exit node and is computed before the scheduling process is begun.

The *Earliest Time First* (ETF) [14] algorithm exhaustively computes the earliest start time of each ready-vertex—processor pair and greedily schedules the ready vertex with the earliest possible start time first.

As depicted in Figure 2(a), the modified HLEFT and ETF algorithms produce identical schedules for the DAG in Figure 1 (although the individual vertices and edges are allocated to processors in a slightly different order). In Figure 2, rows represent parallel processor nodes and columns represent time units. Each row is divided horizontally into two halves. Gray and hatched rectangles represent vertex and edge allocations to the corresponding parallel node's processor and network link, respectively. Absence of rectangles indicates that the processor or network link is idle.



(a) Schedule created by the modified HLEF and ETF algorithms when (v_4, v_6) is considered before (v_3, v_6) while scheduling v_6.



(b) Schedule created by the modified HLEF and ETF algorithms when (v_3, v_6) is considered before (v_4, v_6) while scheduling v_6.

Figure 2: Schedules produced by the modified HLEFT and ETF algorithms for the DAG in Figure 1.

Edges originating and terminating within the same processor do not consume network resources, and therefore, are not shown. Also, note that the multicast edge (v_0, v_3) originates from p_1 rather than p_0 even though v_0 is scheduled at p_0. This is because p_1 has previously received the multicast data in the form of edge (v_0, v_2) and can transmit it to p_3 while p_0 is busy with edge (v_0, v_4).

3.2 Genetic List Scheduling

GAs are optimization algorithms that are based on the principles of natural evolution. A GA iteratively evolves improved solutions by randomly exploring the solution space while simultaneously exploiting the features of the previously discovered solutions. Essentially, the GA optimizes an objective function by repeatedly selecting the "fitter" solutions (as determined by the objective function) to contribute characteristics to their progeny and by discarding the relatively poor solutions. This causes the features of the fittest solutions to remain in the population and repeatedly recombine, producing even better solutions, while the features of poor solutions disappear over time. In GLS, the LS algorithm schedules the ready vertices in the order determined by the GA and solution fitness is derived from the makespan of the associated schedules.

Several genetic operators such as selection, crossover, and mutation are used to combine features from parent chromosomes into offspring chromosomes. The selection operator is responsible for selecting fitter chromosomes for recombination into new chromosomes and for discarding poor chromosomes to make room for the new chromosomes. The crossover operator is used to combine the features of two chromosomes to form a new chromosome. The mutation operator is used to induce stochastic exploration of the search space and to reduce the occurrence of premature convergence of the GA at a local optimum. Convergence is indicated when most of the chromosomes in the population have similar features and nearly identical fitness values. When the population has converged at a local optimum, the ability of the GA to explore new areas in the search space in order to locate the global optimum is reduced. The wide variety of genetic representation schemes, selection operators, crossover operators, and mutation operators proposed in the literature precludes their detailed description here.

4 Approach

The outline of the steady state GLS algorithm investigated in this paper is as follows:

1. Generate the initial population.
2. Use list scheduling to construct a schedule in order to evaluate each of the initial chromosomes.
3. Do while the termination criteria are not satisfied:
    4. Select a genetic operator.
    5. Select chromosome(s) from the local population and apply the operator to produce the offspring chromosome.
    6. Use LS to construct a schedule in order to evaluate the offspring chromosome.
    7. Select chromosome from the local population to be replaced by the offspring chromosome.
8. Use the fittest chromosome to construct the solution schedule.

4.1 Genetic Representation

Most existing LS and GLS algorithms focus on prioritizing vertices in the ready list and can schedule the incoming edges of a vertex in arbitrary order because communication contention is ignored. However, when communication contention is allowed, the order in which edges are scheduled also impacts makespan. For example, the schedule in Figure 2(a) is shorter than the schedule in Figure 2(b) because edge (v_4, v_6) is considered for scheduling before edge (v_3, v_6) while scheduling v_6.

Each chromosome in the GLS developed in this research consists of two vectors of genes. The *vertex vector* contains a gene for each vertex in the DAG and the *edge vector* contains a gene for each edge in the DAG. Essentially, each gene uniquely identifies its corresponding vertex or edge and there are $|V|+|E|$ genes in each chromosome. The position of the vertex and edge genes in their respective vectors determines the priority of the corresponding vertices and edges used by in the LS phase of the GLS.

4.2 Schedule Construction

Given a DAG G and set of parallel nodes P, the LS component of the GLS algorithm essentially operates as described in Section 3.1. The ready list, R, consists of vertices whose predecessor vertices have been scheduled but have not necessarily completed execution. At each step, the ready vertex, $r \in R$, whose gene appears foremost in the vertex vector is removed from the ready list and is allocated to the processor that allows the earliest start time (subject to precedence constraints). Formally, the start time for r is

$$\tau_s(r) = \min\{\tau_{alloc}(r, p) : p \in P\},$$

where $\tau_{alloc}(r, p)$ is the earliest time vertex r can be allocated to processor p. $\tau_{alloc}(r, p)$ is determined as follows. Let $A(p)$ represent the list of intervals in the schedule during which vertices are allocated to processor p. Assume that the intervals in $A(p)$ are ordered according to start times. Therefore, for a_k, the k^{th} vertex in $A(p)$, $(\tau_s(a_k), \tau_e(a_k)]$ represents the interval during which a_k is scheduled on p, and $\tau_e(a_k) \le \tau_s(a_{k+1})$. Similarly, let $L(p)$ represent the ordered list of edges allocated to the communication link on p, and $(\tau_s(l_k), \tau_e(l_k)]$ represents the interval during which edge l_k is scheduled on p's link. Further assume that all $A(p)$ and $L(p)$ contain two sentinel intervals $(-\infty, -1]$ and $(\infty, \infty]$.

$\tau_{alloc}(r, p)$ is the beginning the first idle slot in $A(p)$, after all preceding edges of r have been allocated, that can accommodate the compute time for r. Therefore,

$$\tau_{alloc}(r, p) = \min\{\max(\tau_e(a_k), \tau_{ready}(r, p)) : \\ a_k, a_{k+1} \in A(p), \tau_s(a_{k+1}) - \max(\tau_e(a_k), \tau_{ready}(r, p)) \ge t(r)\},$$

where $\tau_{ready}(r, p)$ is the time the last preceding edge of r is completed on p. Assuming that Q_r represents the list of all incoming edges of r,

$$\tau_{ready}(r, p) = \max\{\tau_e(q, p) : q \in Q_r\},$$

where $\tau_e(q, p)$ is the completion time for edge q with the destination vertex scheduled at p. For an as-yet unscheduled point-to-point edge q with a predecessor vertex σ_q that has been scheduled on processor s and has a completion time $\tau_e(\sigma_q)$,

$$\tau_e(q, p) = \lambda(t(q, s, p), s, p, \tau_e(\sigma_q)).$$

The function $\lambda(w, s, p, \alpha)$ returns the time when the communication operation with weight w completes from processor s to processor p and α is the earliest time that the edge is permitted be scheduled. If s and p are the same,

$$\lambda(t(q, s, p), s, p, \tau_e(\sigma_q)) = \tau_e(\sigma_q)$$

because intra-processor communication consumes no time. Otherwise,

$$\lambda(w, s, p, \alpha) = \max(\tau_e(l'_{i-1}), \tau_e(l''_{j-1}), \alpha) + w$$

when the following condition is true:

$$\forall_{l'_i \in L(p)} \forall_{l''_j \in L(s)} \min(\tau_s(l'_i), \tau_s(l''_j)) - \max(\tau_e(l'_{i-1}), \tau_e(l''_{j-1})) \ge w$$

and $\tau_s(l'_{i-1}) \geq \alpha$ and $\tau_s(l''_{j-1}) \geq \alpha$.

When edge q belongs to the set B of edges associated with a multicast group, there are several source processor choices available. Essentially the processor, s, on which the multicast root vertex, υ, has been scheduled and the set of any of the other processes, O, that have received a copy of the multicast data can be used as the edge source. Therefore,

$$\tau_e(q, p) = \min(\lambda(t(q, s, p), s, p, \tau_e(\upsilon)) \cup \{\lambda(t(q, o, p), o, p, \tau_e(o)) : o \in O\}).$$

4.3 Recombination Operators

Two different crossover operators, *ordered crossover* (OX) [7] and *vector crossover* (VX) are used in this GSL algorithm. In OX, a single crossover point is randomly selected in the vertex vector. The sequence of genes prior to the crossover point is copied from the first parent to the front of the vertex vector in the offspring chromosome. The remaining genes in the first parent (*i.e.*, following the crossover point) are copied into the offspring gene in the order they appear in the second parent. The same operation is also performed on the edge vectors to construct the offspring chromosome. In VX, the first parent contributes a complete copy of its vertex vector and the second parent contributes a complete copy of its edge vector to construct the offspring chromosome.

The mutation operator swaps the location of a pair of genes within the vertex vector, the edge vector, or a pair each from both vertex and edge vectors. One of these three mutation options is selected with equal probability.

The GLS produces good results when the probabilities of selecting the OX, VX, and mutation operators are π_{ox} = 0.75, π_{vx} = 0.20, and π_m = 0.05, respectively.

4.4 Selection

Other researchers have observed that making direct use of the objective function (*i.e.*, makespan in this case) to compute the fitness of chromosomes can lead to premature convergence [3]. This occurs when a single chromosome with a significantly better than average objective function value is repeatedly selected for reproduction, leading to reduced genetic diversity in the population. In order to prevent this situation, the rank of chromosomes, rather than their objective function values are used to compute their fitness. The rank of chromosome c is the number of chromosomes in population Ω that produce worse schedules than c. More precisely,

$$rank(c) = |\{c_i \in \Omega : makespan(c) < makespan(c_i) \vee \text{ if } makespan(c) = makespan(c_i) \rightarrow processors(c) < processors(c_i)\}|.$$

In order to further reduce the ability of high ranking individuals to dominate the population, the fitness of a chromosome is made inversely proportional to the number of offspring it has produced. This results in the following fitness function, also used by Grajcar [12]:

$$\varphi(c) = \frac{rank(c)}{|offspring(c)| + 1}.$$

The chromosome with the largest fitness value in a random subset of Ω is selected for reproduction. Similarly, the chromosome with the least fitness value from another random subset of Ω is selected for replacement. In the experiments reported here, selection subset sizes ranging from 2% to 10% of $|\Omega|$ worked well for populations ranging from 200 to 800 chromosomes.

4.5 Parallel Implementation

A parallel implementation of the GLS using the *synchronous connected island model* [10] is also under investigation. In this model, the parallel GLS processes operate on separate local populations and communicate synchronously to exchange the fittest chromosomes with each other periodically. This "migration" of chromosomes enhances the exploitation of high quality solutions in parallel GAs, while the relative isolation of the subpopulation enhances genetic diversity and prevents premature convergence.

The number of iterations between migrations is reduced exponentially to a specified minimum, m (*i.e.*, migrations occur after M, $M/2$, $M/4$, $M/8$, …, $2m$, m, …, m iterations). The value of M is tuned so that high-frequency migrations occur shortly before the number of iterations expected to produce results, or more realistically, the maximum allowed iterations or time elapses. This strategy emphasizes exploration of the solution space at the beginning of the evolutionary process and emphasizes exploitation of good genetic information towards the latter stages.

As suggested in [19], varying the GA control parameters in each of the N parallel GLS processes can lead to increased genetic diversity. Therefore, the mutation, vector crossover, and ordered crossover rates used in a parallel GLS process $n_i \in \{0, 1, …, N - 1\}$ are given by

$$\pi_m(n_i) = 0.05 + 0.05 \times n_i/N,$$
$$\pi_{vx}(n_i) = 0.20 + 0.10 \times n_i/N, \text{ and}$$
$$\pi_{xo}(n_i) = 0.75 - 0.15 \times n_i/N,$$

respectively. This implies that the mutation rate ranges between 0.50 and 0.10, the feature crossover rate ranges between 0.20 and 0.30, and the order crossover rate varies between 0.75 and 0.60. These ranges appear to work well for the sample scheduling problems solved in the research effort.

5 Experimental Analysis

5.1 Setup

DAGs with three distinct types of structures are used in order to evaluate the performance of the GLS algorithm. The first type of DAG has an "out-tree" structure as illustrated in Figure 3(a). The number of outgoing edges from each vertex is randomly selected from a uniform distribution between one and ten. The second type of DAG has a "fork-join" structure illustrated in Figure 3(b). The width of each "fork" in the fork-join DAGs is selected randomly from a uniform distribution between two and ten. The third type of DAG has a random structure. The random DAG has eight times as many edges as vertices connecting randomly selected edges (*i.e.*, on average, each vertex has eight incident or outgoing edges).

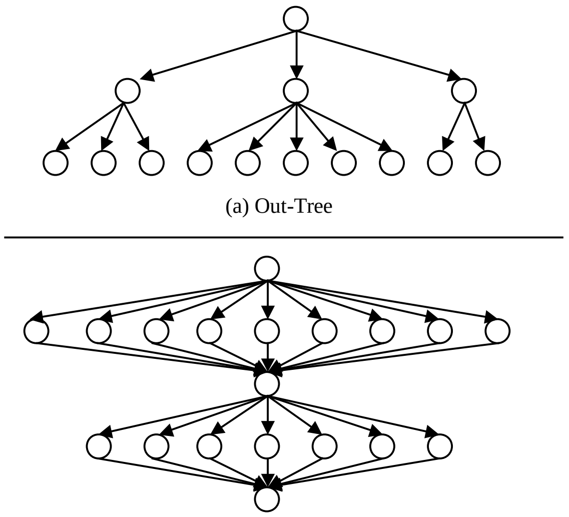


Figure 3: Sample DAG types

Six instances of each type of DAG were created. Each of these DAG instances consists of approximately 300 vertices. Within a DAG, each vertex is assigned a weight selected from a normal distribution with mean = 10 and variance = 3. Edge weights are also selected from normal distributions with a variance = 3. However, the mean of the edge distribution is set to 5, 10, or 20 at the beginning of the generation process. Additionally, in each DAG, either no edge is part of a multicast group, or all outgoing edges from every vertex are grouped into multicast groups (note that edges originating at different vertices cannot belong to the same multicast group). These combinations of structure, edge weight distributions, and multicast or non-multicast edges result in a total of 18 distinct DAGs.

In order to provide a basis for comparison, the HLEFT and ETF algorithms were also modified to handle multicast edges and communication contention. Nominally, HLEFT and ETF do not consider edge ordering when scheduling. However, because edge ordering is significant in the presence of communication contention, all inbound edges of a vertex are scheduled in descending order by weight in the modified HLEFT and ETF algorithms. While this "first fit decreasing" bin packing heuristic results in the shorter schedule of Figure 2(a), it does not guarantee optimality. Computational costliness precluded the exhaustive examination of all possible edge orderings in this research effort.

5.2 Sequential Results

A problem with GAs is that the number of iterations required to produce near-optimal results is not known *a-priori*. Indeed, for many large NP-hard problems, the precise nature of the optimal solution is unknown, and therefore, cannot be used to determine the termination point for the GA. Because of these reasons, GAs are typically executed for a fixed number of iterations or for a fixed amount of time.

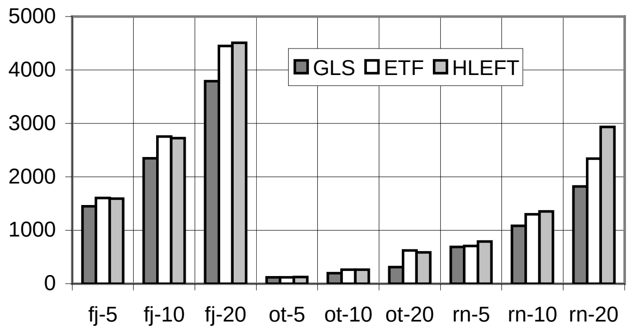


Figure 4: Schedule makespans for the multicast DAGs

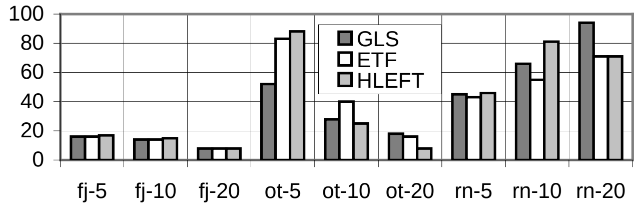


Figure 5: Schedule processors for the multicast DAGs

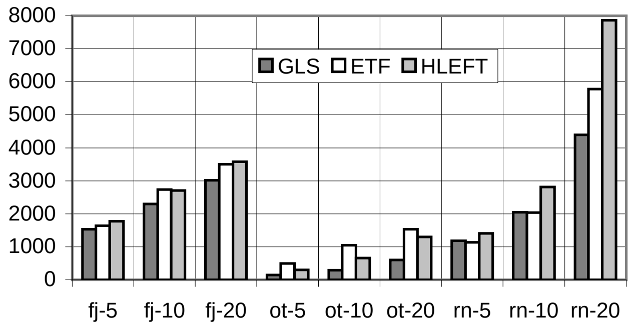


Figure 6: Schedule makespans for the pt-to-pt DAGs

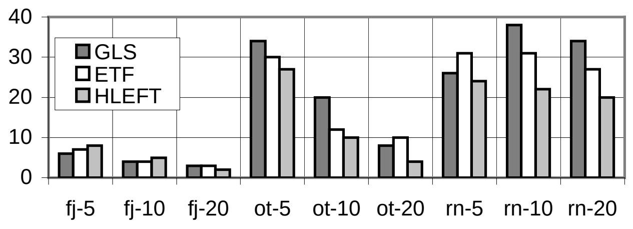


Figure 7: Schedule processors for the pt-to-pt DAGs

The large number of edges in the random DAG instances causes the GLS implementation to take a long time to construct schedules. Therefore, each GLS run was initially limited to a maximum of 9,000 iterations in order to complete the experiments in a reasonable amount of time. Also, based on results from preliminary experimentation, the population size for each GLS run was set to 600. Because of the stochastic nature of GAs, the GLS algorithm was applied on each DAG instance at least three times and the median results are reported here.

Figures 4 and 5 illustrate the makespan and number of processors, respectively, in the schedules for the nine DAGs with multicast operations. Figures 6 and 7 illustrate the makespan and number of processors, respectively, in the schedules for the nine DAGs with only point-to-point communication. In order to determine the shortest possible schedules could be produced by this GLS, the GLS was permitted to use an unlimited number of processors in the schedules.

Labels fj-*x*, ot-*x*, and rn-*x* along the bottom axis of the figures indicate the results for the fork-join, out-tree, and random DAGs, respectively. The *x* value indicates the mean edge weight in the DAG instances.

In terms of minimizing makespan for the 18 DAGs, GLS outperformed HLEFT in all instances and ETF in 15 instances. ETF did better than HLEFT in 11 instances. When the three instances for which the GLS was unable to find better makespans than ETF were re-executed without iteration restrictions, the GLS took 1,537, 3,919, and 16,464 iterations to produce shorter schedules than ETF for multicast-ot-5, pt-to-pt-rn-5, and pt-to-pt-rn-10, respectively. This highlights the problem with the probabilistic nature of GAs in that it took several attempts before the GLS was able to solve these three problems.

In general, shorter schedules should require a greater number of processors than longer schedules for the same DAG instance. However, of the 15 instances in which GLS produced shorter schedules than ETF, GLS used equal or smaller number of processors than ETF in 8 instances. Similarly, in 8 of 18 instances, GLS used fewer processors than HLEFT. This ambiguity in processor utilization results suggests that the strategy of minimizing the start time for vertices at the cost of allocating a new processor is non-optimal. This is further illustrated by the schedule in Figure 8 for the DAG in Figure 1. This schedule requires only three processors. However, none of the LS algorithms tested is able to produce this schedule.

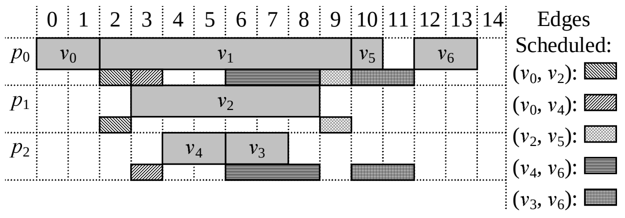


Figure 8: Optimal schedule for the DAG in Figure 1

5.3 Parallel Performance

In order to evaluate the performance benefits of parallelizing GLS, the time required by the sequential and parallel GLS implementations to evolve schedules with specific makespans is measured and compared. Speedup is given by $S = T_s/T_p$ where T_s is the time taken by the sequential implementation and T_p is the time taken by the parallel implementation. As has been observed by other researchers (*e.g.*, [13]), this strategy for computing speedup is required in order to accurately account for the benefit of using larger global population sizes and improved population diversity of parallel GAs.

Table 1 shows the speedup achieved by the MPI-based parallel GLS implementation executed on a 8-node dual 450MHz PIII cluster with a high-speed Giganet interconnect fabric. The multicast fj-10 DAG instance was used for this analysis because this DAG's structure allows a large number of iterations to complete in a relatively short amount of time. This enabled the use of challenging makespan thresholds, requiring over 50,000 iterations. Furthermore, the global population size of 9,600 was used in this experiment (*i.e.*, the local population size for each parallel GLS process was 9600/*P*, where *P* is the number of processes in the parallel GLS). Several runs of the sequential and parallel GLS implementations were performed to collect execution time data and the best results were used to compute speedup.

	Seq	P=2	P=4	P=8	P=16
Speedup		0.98	2.20	4.07	3.43
Makespan range after 80K iterations	2359-2325	2329-2290	2286-2266	2290-2232	2286-2242

Table 1: Performance of Parallel GLS

Although, these speedup figures in Table 1 are small, an analysis of the schedule makespans that resulted when each algorithm was run for 80,000 iterations demonstrates that given a fixed amount of time, the parallel version running on 8 or 16 processors evolves better schedules than the sequential GLS. The second row of Table 1 lists the maximum and minimum schedule lengths that resulted for the sequential and the various parallel configurations of the GLS.

6 Conclusions

The research presented in this paper extends existing genetic list scheduling research by considering communication contention at the processor-to-network links. This communication model more accurately reflects the behavior of processor-to-network links in the switched network technology used to deploy clusters. Therefore, the resulting schedules can be more readily realized in the scheduling of parallel applications on practical parallel systems.

In order to construct efficient schedules in the presence of communication link contention, the GA approach used in this research seeks to optimize both vertex and edge priorities for the list scheduler. Optimizing edge priorities is important because the order in which communication links are allocated to edges affects schedule makespan in the presence of communication contention. The GLS approach developed here also seeks to efficiently schedule multicast operations in the presence of communication contention.

Because of the NP-hard nature of the DAG scheduling problems, optimal solutions were not available for the large problems addressed here. However, the comparison of schedules resulting from GLS, HLEFT and ETF algorithms suggests that GLS is able to produce much better results at the cost of using significantly more time.

The probabilistic nature of GLS means that the algorithm may need to be run several times in order to find a near-optimal solution. A related problem is that it is difficult to predict how long GLS must run before a near-optimal schedule is evolved.

As expected, the parallel implementation of GLS produced better schedules in a shorter amount of time than the sequential GLS. The speedup characteristics of this parallel implementation, based on the synchronous connected island model, were disappointing. However, the parallel GLS was able to delay premature convergence and, therefore, was able to discover schedules shorter than the shortest schedule produced by the sequential GLS.

Ongoing research is focused on extending this work to construct schedules for heterogeneous environments with multiple communication links per processor. The feasibility of applying GLS in scheduling for real-time systems is also under investigation as is the optimization of the parallel GLS control parameters (*e.g.*, population size, frequency and topology of migrations, adaptive crossover and mutation rates, and termination criteria).

7 References

[1] T. L. Adam, K. M. Chandy, and J. Dickson, "A Comparison of List Scheduling for Parallel Processing Systems," *Communications of the ACM*, vol. 17, December, pp. 685-690, 1974.

[2] I. Ahmad, Y. Kwok, and M. Wu, "Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors," in *Proc. of the International Symposium on Parallel Architectures, Algorithms, and Networks*, pp. 207-213, 1996.

[3] T. Bäck and F. Hoffmeister, "Extended Selection Mechanisms in Genetic Algorithms," In Proc. of the 4th International Conference on Genetic Algorithms, pp. 92-99, 1991.

[4] M. Barnett, D. Payne, and R. van de Geijn, "Optimal Broadcasting in Mesh Connected Architectures," TR-91-38, University of Texas, 1991.

[5] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, John Wiley and Sons, New York, 1976.

[6] W. W. Chu, M. Lan, and J. Hellerstein, "Estimation of Intermodule Communications (ICM) and its Application in Distributed Processing Systems," *IEEE Transactions on Computers*, vol. C-33, no. 8, pp. 691-699, 1984.

[7] L. Davis, "Applying Adaptive Algorithms to Epistatic Domains," in *Proc. of the 9th international joint conference on artificial intelligence*, pp. 162-164, 1985.

[8] D. Gajski and J. Peir, "Essential Issues in Multiprocessors," *IEEE Computer*, vol. 18, no. 6, 1985.

[9] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp 276-291, 1992.

[10] V. S. Gordon and D. Whitley, "Serial and Parallel Genetic Algorithms as Function Optimizers," Technical Report CS-93-114, Colorado State University, 1993.

[11] M. Grajcar, "Strengths and Weaknesses of Genetic List Scheduling for Heterogeneous Systems," in *Proc. of the 2nd International Conference On Application of Concurrency to System Design, IACSD*, 2001.

[12] M. Grajcar, "Genetic List Scheduling Algorithm for Scheduling and Allocation on a Loosely Coupled Heterogeneous Multiprocessor System," in *Proc. of the 36th Design Automation Conference*, pp 280-285, 1999.

[13] W. E. Hart, S. Baden, R. K. Belew, and S. Kohn, "Analysis of the Numerical Effects of Parallelism on a Parallel Genetic Algorithm," in *Proc. of the 10th International Parallel Processing Symposium*, pp. 606-612, 1996.

[14] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM Journal of Computing*, vol. 18, no. 2, pp. 244–257, 1989.

[15] Y. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406-471, 1999.

[16] Y. Kwok and I. Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors using a Parallel Genetic Algorithm," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 58-77, 1997.

[17] Y. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506-521, 1996.

[18] MPI/RT Forum, *Document for the Real-Time Message Passing Interface (MPI/RT-1.1)*. http://www.mpirt.org, 2001.

[19] J. C. Potts, T. D. Giddens, and S. B. Yadav, "The Development and Evolution of an Improved Genetic Algorithm Based on Migration and Artificial Selection," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 1, pp. 73-86, 1994.