A Transformation-Based Approach to Static Multiprocessor Scheduling

Alan Sheahan and Conor Ryan
Biocomputing and Developmental Systems Group
Computer Science and Information Systems Department
University of Limerick, Ireland
Alan.Sheahan@lit.ie, Conor.Ryan@ul.ie

ABSTRACT

This paper describes a novel Genetic Algorithm (GA) approach to scheduling. Although the particular problems examined are all multi-processor scheduling types it can, because the algorithm takes a DAG (Directed Acyclic Graph) as input, be applied to any scheduling problem represented by a DAG.

The algorithm works by calculating the *mobility* of each node in the graph and using this to constrain the search space in a useful way, that is, nodes can be scheduled using a larger range of *levels* in the final schedule than those obtained by a simple levelling of the DAG.

The GA itself operates by evolving sequences of transformations which build up ever increasing lists of task associations, using two simple transformations. We show that our algorithm can outperform standard methods, both traditional and GA based, at considerably lower costs.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Genetic Algorithms*

General Terms

Algorithms, Performance

Keywords

Scheduling, Multi-processor, graph drawing

1. INTRODUCTION

Scheduling is one of the earliest applications of computers to problem solving. It is a hard, NP complete problem which can be applicable in all sorts of different domains. This work is concerned in particular with multi-processor scheduling, in the form of producing a list of **processor/task** pairs from a Directed Acyclic Graph (DAG). However, DAGs are general representations of task dependencies, and so can be used for different kinds of schedules.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-130-9/08/07…$5.00.

In recent years, it has become feasible to use Genetic Algorithms to produce schedules. The advantage GA approaches have is their parallel search. Traditional approaches to scheduling are virtually all based on greedy (to various extents) heuristics and are not always capable or even likely to find optimal schedules. This is acceptable in many scheduling applications where it is more important to produce a *good* schedule *quickly* than it is to produce an optimal one. In "once and for all" schedules, where a schedule is to be used many times, this is less acceptable.

GA approaches are, theoretically, less likely to be to trapped by the sort of local optima greedy heuristics can fall prey to by virtue of their parallel, population-based search. One difficulty with GAs, however, is the brittleness of the individuals, and, particularly with an absolute representation, a crossover between two good schedules can produce non-viable offspring.

This paper describes a representation in which the genome comprises sequences of transformations which cluster tasks together and move them around a virtual parallel machine. Crucial to this is a notion of **mobility**, information about how tasks can be moved. We describe how this information can automatically be generated from each DAG.

We show that highly efficient and sometimes optimal solutions can be obtained from only 2 such types of transformations on the genome. We demonstrate a high performance across a wide variety of diverse benchmark graphs in the literature, when compared with other GA and traditional methods. In addition, a superior *average* performance is achieved when contrasted with a well-known GA approach to the problem, thus showing the robust nature of the system. This superior performance is achieved at a much smaller computational cost (identical population sizes running for 50 instead of 3000 generations).

The paper is organized as follows. Section 2 describes the problem in terms of DAGs and section 3 details traditional approaches to scheduling. We describe our approach to the problem in sections 4 and 5. Sections 6 and 7 describe the experimental set up and section 8 shows our results.

2. PROBLEM DEFINITION AND ASSUMPTIONS

We have assumed a well-accepted model of multiprocessor systems where parallel programs are modelled by Directed Acyclic Graphs (DAGs) [5], [1], [2], [4]. The DAG is defined by a tuple G = (V, E, C, T), where V is the set of graph nodes, E corresponds to the set of edges, C represents the set of edge weights and T is the set of node weights.

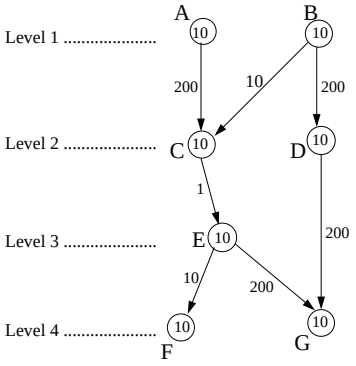

Figure 1: A typical DAG

Figure 1 shows a typical DAG, where each node corresponds to a set of one or more instructions that are executed to completion on the same processor without interruption. The node weight is the time necessary to complete the execution. Each edge, joining a source and destination node, represents a precedence constraint imposed on the final schedule such that the source node must complete its execution and communicate the necessary information to the destination node before the destination node can begin execution. The edge weight is the time delay incurred in the communication across the network. If the two nodes are scheduled on the same processor, the communication cost is assumed zero due to the relatively insignificant amount of time to retrieve information from local memory compared to retrieval across a network. Note also that the 7 nodes in Figure 1 occupy 4 distinct levels, where the level that a node occupies is one greater than the largest level of its immediate predecessors [3]. All entry nodes (nodes A and B in Fig. 1) have level 1. The concept of levelling is central to the system presented here.

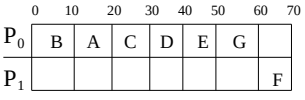

Figure 2: A sample schedule for the DAG in Figure 1

Figure 2 depicts a schedule on a 2-processor machine. Notice that the communication delay represented by the edge weight from task A to C does not apply as the tasks reside on the same processor, whereas the edge joining task E and F delays the execution of the latter by 10 time units due to the fact that the tasks reside on different processors.

The model assumes that the target architecture comprises a fixed number of identical processors which are completely connected via a uniform network (i.e. a network where the communication delay between any pair of processors is the same). Nodes (also known as tasks) may send and receive data in parallel and may multicast information to more than one receiver. It is also assumed that communication is asynchronous and can occur simultaneously with computation.

Static task graph scheduling involves the assignment of tasks to processors, the ordering of the tasks on each processor and the determination of each task's execution start time in order to achieve a specified objective [7]. While many objectives exist (e.g. minimise processor utilisation, balance workload across all processors etc..), the generally-accepted primary goal (and the only one under consideration here) is to minimise the overall program execution time, known as the *Parallel Time* (PT) [4], [1], [2]. This problem has been shown, except under simplifying assumptions, to be NP-Complete [6].

Some techniques allow the redundant allocation of multiple instances of tasks to processors known as *Task Duplication* in order to reduce the PT further [2]. While the concept of Task Duplication is central to these systems, the system presented is the only one of its kind to incorporate task duplication as an option.

3. TRADITIONAL APPROACHES

The earliest approaches to solving this problem were deterministic by nature, the most common of which comprised the following techniques:

List Scheduling (LS) algorithms schedule nodes according to a priority-assignment scheme, where higher priorities are given to nodes based on some graph/schedule characteristic e.g. nodes whose predecessors are already scheduled or nodes whose in-coming edges are heavily weighted etc.

Critical Path (CP) heuristics schedule nodes by trying to shorten the longest execution path (critical path) in the DAG. This is done by incrementally zeroing critical path edges and consequently clustering nodes together for execution on the same processor.

Graph Decomposition (GD) methods parse the DAG into a hierarchical tree of subgraphs and subsequently focus on subsections of the graph that are independent and can be parallelised.

While these methods can produce reasonably good schedules, there are a number of problems associated with them. These include:

(i) Ties between scheduling decisions are invariably broken arbitrarily.

(ii) The algorithms are greedy, making choices based on localised information.

(iii) These algorithms only perform reasonably well when the ratio of edge weights to node weights is relatively small (i.e. coarse grained graphs) and often produce schedule lengths in excess of a serial execution where the cost of communication is relatively high (fine grained graphs) [1].

Since then, significant improvements have been made in this area using evolutionary approaches which have led specifically to a reduction in the overall schedule time. In particular *Genetic Algorithms* (GAs) have been applied by either combining the GA with list scheduling techniques to find favourable priority-assignment rules or using the GA to explicitly carry out the actual assignment and ordering of the tasks on the processors [2]. The system presented here adopts the latter form.

4. DESIGN CONSIDERATIONS

The limitations of the traditional methods mentioned above motivated the identification and exploration of the main characteristics of the problem so that a suitable representation could be found that would minimise the search space and tailor the search to exploring only:

(i) schedules that are likely to be *efficient* (schedules with a speedup greater than 1) and

(ii) legal, in terms of adhering to the precedence constraints of the DAG.

To tailor the search for efficient schedules, a mechanism based solely on DAG information is incorporated into the mapping from the genome to schedules. In particular, independent tasks (i.e. where neither task is a predecessor of the other) should be capable of being scheduled in any order on the same processor, however such tasks with level values that differ greatly are unlikely to produce efficient schedules by scheduling them in reverse topological order. Figure 3 shows a DAG and an optimal 2-processor schedule.

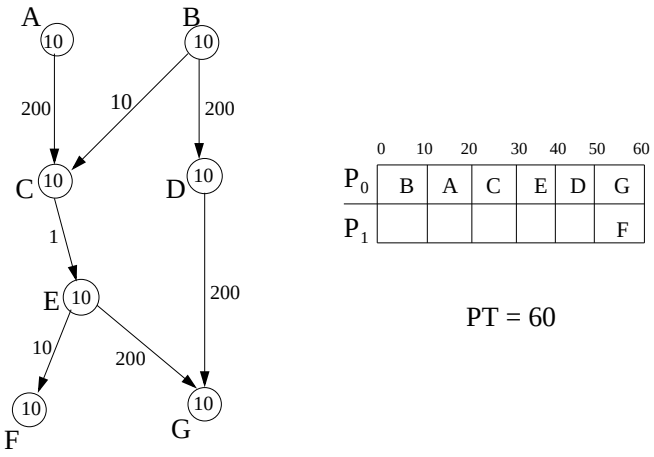

Figure 3: DAG from Figure 1 and optimal schedule. This differs from the schedule in Figure 2 by reversing tasks D and E.

Critical to the optimality of the schedule presented here is the fact that task E is scheduled before task D on the same processor (as task F must wait 10 time units after E is finished), despite the fact that task E is situated a level below task D in the DAG. As a result it was felt that a system capable of constructing schedules with this characteristic should be investigated. However, it should be noted that checking all possible configurations of schedules involving independent tasks is unnecessary as many produce inefficient schedules e.g. scheduling task F before task D on the same processor will produce an inefficient schedule. The system presented here gives the GA room to explore the scheduling of sensible configurations of independent tasks on the same processor and unlike the typical deterministic approaches, is not reliant on localised information in making such choices.

To achieve this, a **bi-levelling** procedure on the input DAG is presented, identifying with each node, an upper and lower bound on the level that the node can exist on. Consequently, nodes are perceived to possess vertical mobility in a discrete level sense and hence, are not restricted to occupying a single level. The system presented orders the execution of the tasks on each processor on the basis of task level, but the introduction of a level mobility for the nodes allows for a certain amount of flexibility in terms of which tasks may be scheduled before others, assuming there are no dependencies between them (achieved by the system always adhering to the precedence constraints of the DAG. The bi-levelling is carried out as follows:

First, a **Down levelling** is applied to the graph whereby a lower bound is acquired for each node level [Figure 4]. This is initialised by labelling *entry nodes* (nodes of in-degree zero i.e. nodes A and B) with level one. All nodes directly connected to these with an edge, i.e. immediate successors, are (at least temporarily) assigned a level value of two. This is repeated recursively until all nodes have been visited in turn and assigned a permanent level. The temporary nature of the assignment is due to the fact that some nodes' level values will change as the graph is traversed due to the presence of *long edges* (edges that span more than one level) e.g. task G would initially be assigned a temporary level value of 3, due to its incoming edge from D, however this is a long edge because task E (on level 3) is also its predecessor. Hence, G is assigned a permanent level of 4.

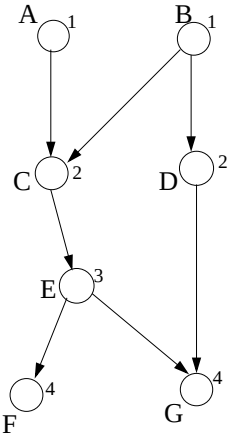

Figure 4: *Down* Levelling

A subsequent **Up levelling** is also carried out to acquire an *Initial* set of task level values in a similar manner to the Down levelling but this time starting with the *exit nodes* (nodes of out-degree zero i.e. nodes F and G) and working upwards. Now, a tranformation of these Initial level values is performed which produces a set of *Final* Up levels [Figure 5], which represent a set of upper bounds on the nodes' levels. The tranformation is obtained by the following mapping:

$$Final\_uplevel[i] = (num\_levels + 1) - Initial\_uplevel[i]$$

Here, num_levels refers to the total number of distinct levels in the graph (e.g. 4 levels).

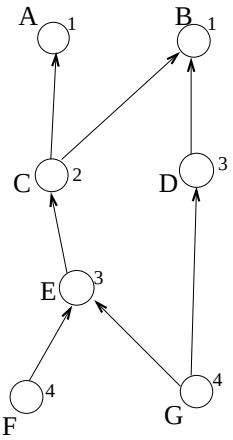

Figure 5: *Up* Levelling: Final

The two sets of levels generated are combined in Figure 6. They independently reveal a levelling that is consistent with the precedence constraints of the DAG and depict a sense of how vertically mobile or unconstrained a node is in a level sense within the graph. Here, task D can be seen objectively to exist on either level 2 or 3 while task E can only exist on level 3. This has significance when it comes to ordering the two tasks on the same processor. Because both tasks can exist on the same level, the system presented makes no distinction as to which must be scheduled before the other. As we saw earlier, for optimality task E must be scheduled before task D [Figure 2]. However, the inefficient scheduling of task F before task D on the same processor is impossible as the tasks have no levels in common.

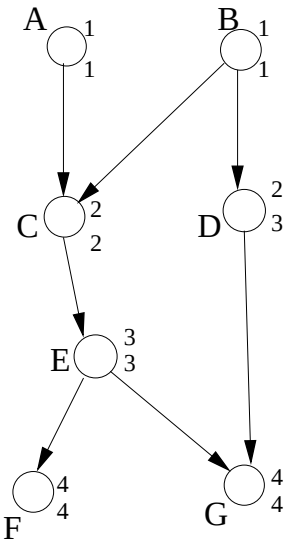

Figure 6: Bi-levelled DAG

This characteristic of the system tailors the search to the exploration of sensible schedules, dependent on extracting information based solely on the structure of the graph, which subsequently provides a platform on which the GA can potentially optimise incorporating graph node and edge weight specifics. To ensure the production of legal schedules, a deadlock-detection mechanism was incorporated into the system. Deadlock may occur in two ways:

(a) Intra-processor deadlock occurs where the scheduling of two dependent tasks on the same processor conflicts with the precedence constraints of the DAG. Figure 7 shows a different bi-levelled DAG. As a result of the bi-levelling procedure carried out, it can be seen that tasks A and C can exist on common levels 3 and 4. Because of this overlap, the system presented here is capable of scheduling the tasks in any order, which will clearly lead to deadlock if task C is scheduled before A on the same processor. This is avoided however, by deploying a **Deadlock detection mechanism** for each individual before fitness evaluation. The procedure checks each pair of tasks on each processor in the final schedule for DAG precedence violations and swaps the positions of any pair found to be non-compliant. This guarantees that all individuals with deadlock of this nature still get to contribute to the evolutionary process.

(b) Inter-processor deadlock occurs where the arrangement of tasks between two or more processors makes the

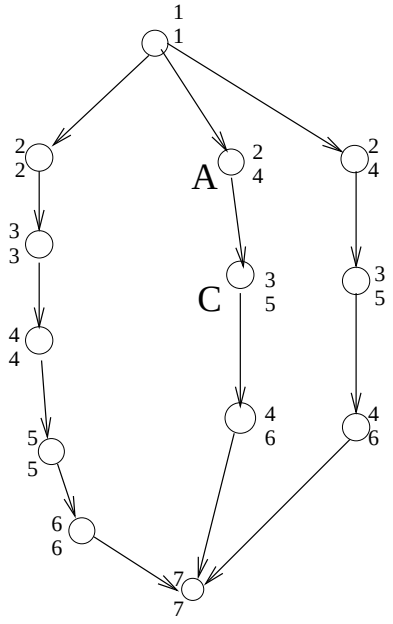

Figure 7: Intra-processor Deadlock

schedule invalid. Figure 8 shows a schedule for the DAG in Figure 7 that displays this type of deadlock. Here, task C must be executed before task B on one processor and at the same time task D is inhibiting task A's execution on another processor. While no processor in isolation has violated the precedence constraints of the DAG, clearly all four tasks are mutually inhibited from executing. This type of deadlock has extremely low incidence (0.04 percent) and therefore individuals with such a characteristic are given a fitness of zero and killed off.

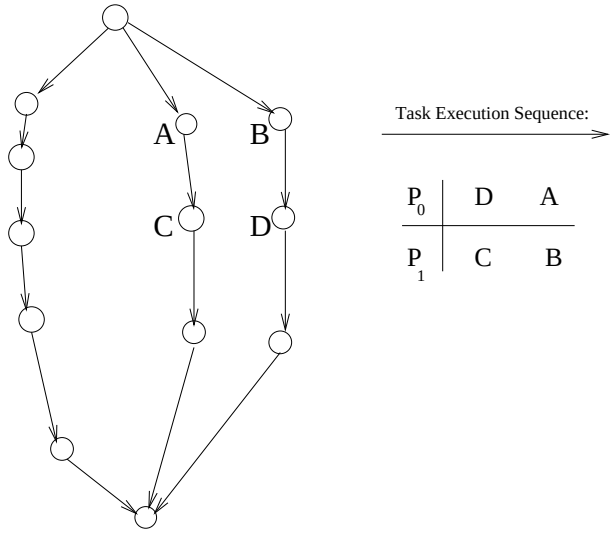

Figure 8: Inter-processor Deadlock

Some schedules can benefit from the allocation of multiple instances of certain tasks to a number of processors in an effort to further reduce the PT. This is known as *Task Duplication*. Ordinarily, the communication delay between dependent tasks that are scheduled on different processors,

dictated by the weight of the connecting edge in the DAG, must be taken into account. However, by creating duplicates of the source task (that distributes the data) on different processors, the destination tasks (that receive the data) are not required to wait since all the data is retrieved from local memory. It should be noted that duplicates must also receive data from their predecessors (where they exist) residing on different processors and can therefore delay the schedule. Figure 9 shows (a) a DAG (b) an optimal 2-processor schedule without duplication and (c) a schedule benefitting from duplication of task C.

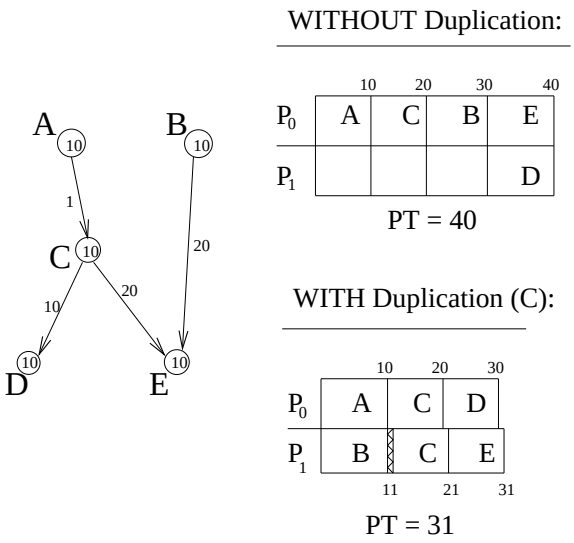

Figure 9: Duplication Benefitting a Schedule

Note that the duplication of task C means that the edges CD and CE have effectively been zeroed because the data generated from the execution of task C is available in local memory of both processors. In addition, despite the fact that task C must wait one time unit on processor 1 for the data to arrive from task A, it still results in an overall reduction in the PT. While duplication can potentially reduce the Parallel Time, it is possible for it to increase it significantly also. Figure 10 displays the scheduling effects of a poor choice of task (task E) to duplicate as it must wait 20 time units for the data from task B before it can execute on processor 1.

While there may be many factors that determine the suitability of a task for duplication, it was decided to favour those with a larger ratio of outgoing edges to incoming edges. In Figures 9 and 10 above, task C has more outgoing than incoming edges compared with task E and therefore it is more likely to be a better choice of task to duplicate. Furthermore, duplication rates (where it is applied) are kept quite small as duplication increases the search space for the GA considerably.

5. REPRESENTATION

This section describes a representation that exploits the information that the methods in the previous section extract from the DAG. Specifically, the levelling information is used in two ways. First, to specify a sensible range of possible scheduling choices for each task, and second, to order the tasks on each processor. This is necessary because

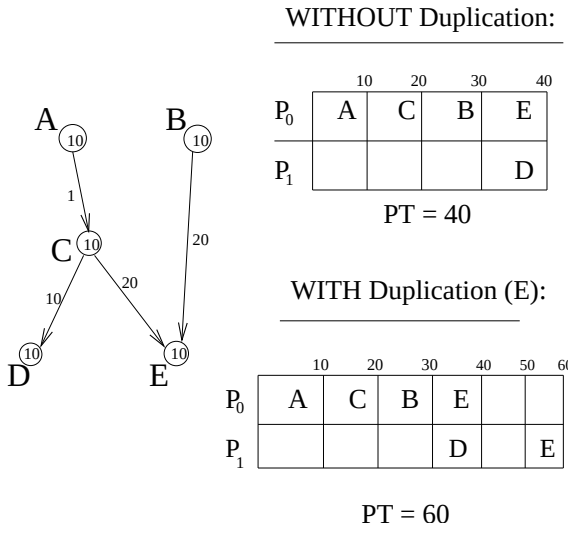

Figure 10: Duplication Inhibiting a Schedule

rather than evolving absolute schedules, we instead evolve sequences of *grabs* that cluster the tasks on the various processors.

The genome is divided into two parts [Fig. 11], one dealing with the level information and the other dealing with the association of tasks to processors and to other tasks. The chromosome uses the level information to map the tasks onto a Virtual Parallel Machine (VPM). At this stage, information such as Parallel Time can be measured.

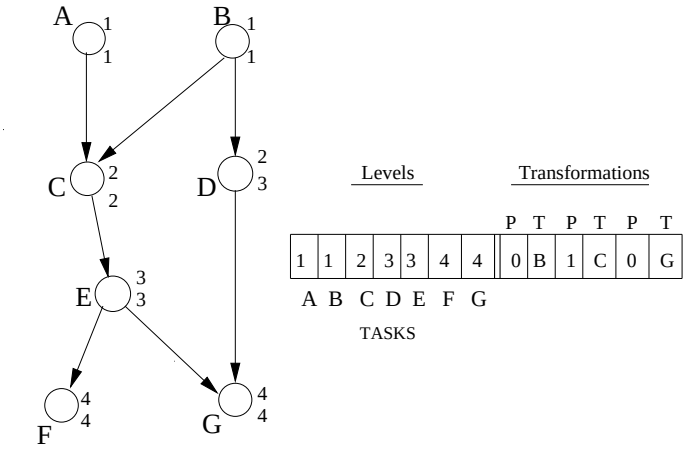

Figure 11: Genome Structure

(i) The first section is of fixed length and comprises a list of level values, one for each node in the DAG. These are generated randomly from within each node's level range for all individuals in the first generation (as dictated by the bi-levelling procedure referred to earlier). Each chromosome is generated such that there is a lower bound on its length. This minimum length comprises the first section of the chromosome.

(ii) The second section consists of none or more pairs of integers which represent Processor-Task transformations. These transformations dictate how tasks are moved about between processors on the VPM. Once the genome has been

read, the configuration of tasks that results constitutes that particular individual's schedule.

The key here is that only two transformations are necessary to produce valid and highly efficient schedules. Where task duplication is opted for, one extra transformation suffices.

5.1 Task List Construction

Initially, all tasks are attached to the first processor on the VPM. This creates what is referred to as the processor's *task list*. Transformations present in the second section manipulate the location of the tasks so that the final configuration determines which processor each task is to be executed on. The first and generally more abundant form of transformation, referred to as a *grab* is where the designated task is removed from the processor on which it is currently located and is appended to the last task of the task list of the specified (destination) processor. This provides a chronologically constructed list of tasks for each processor. Figure 12(a) shows 2 partial task lists, constructed from the transformations of the sample genome sub-section (included in the figure).

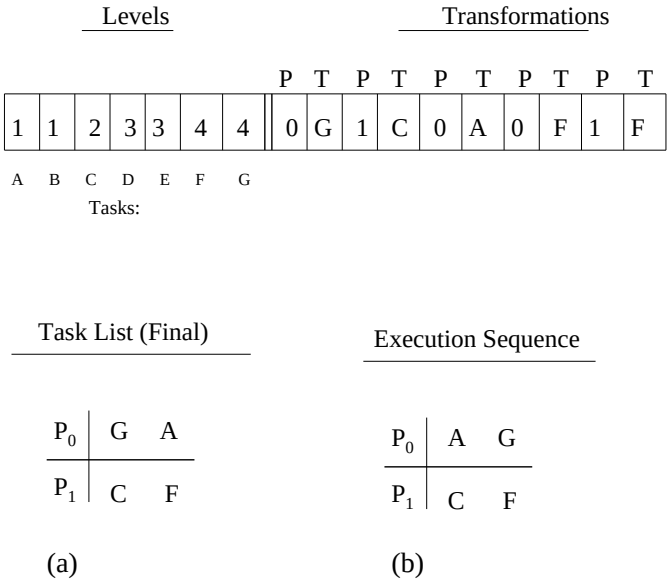

Figure 12: Sample partial genome resulting in a partial schedule

5.2 Task Ordering

This time-ordered list of tasks has two functions: The first determines the *execution sequence* of the tasks previously assigned to each processor. The sequencing is based on the level information from the first section of the genome. Figure 12(b) shows the partial execution sequences arising from the genome levelling information being applied to each processor's task list. Processors are taken in isolation and each task is taken in turn from the processor's task list (starting with the earliest task) and scheduled on the basis of level (i.e. each task is scheduled directly before the earliest task on the list with greater level. If there are no tasks present, it is scheduled at the start of the sequence, whereas if there are no tasks present with greater level, it is scheduled at the end of the sequence). Each processor's execution sequence represents the final order of execution of its tasks. The schedule produced is then checked for deadlock and corrected where necessary resulting in a phenotype whose fitness is evaluated.

Note that Processor 0's execution sequence requires that task A be executed before task G because it has a lower level value, despite A being acquired by the processor with an earlier transformation. Note also that task F is acquired by Processor 0 initially but subsequently moved to Processor 1 in the final configuration. The second function of the list of chronologically acquired tasks is central to the role of the remaining transformation.

5.3 Task Clustering

The second transformation, referred to as a *cluster*, whose rate of occurrence is controlled by an external parameter, is identified by a special task value on the genome and applies solely to the processor gene which is situated just prior (due to the processor-task pairing). This specifies (using the chronologically ordered list of tasks) that the processor, where applicable, clusters its two most recently acquired tasks together. The coalition is treated in exactly the same manner as a single task entity for the remaining set of transformations for the individual. In particular, if a task that is grouped in this way is to be relocated by a subsequent *grab* transformation on the genome, all tasks belonging to its cluster are relocated too. In addition, cluster sizes may be increased for an individual by assimilating other tasks or indeed other clusters as dictated by subsequent clustering genes on the genome. In all cases, the transformations maintain genome integrity by preserving the chronology of all clustered tasks.

The idea here is to encourage the formation of subsets of tasks that work well together on the same processor that the GA can exploit.

5.4 Task Duplication

The final transformation is only necessary where task duplication is desirable. Similar to the *cluster* transformation above, the rate of occurrence of this *duplicate* transformation is also controlled by an external parameter but this time it is identified by a special processor value on the genome and therefore applies only to tasks. The task value that immediately follows is ignored because the duplication mechanism presented must specify a processor value. This *duplicate* transformation overrules the next transformation on the genome (providing it is a *grab*) by duplicating the specified task on the designated processor (i.e. similar to the standard *grab* except it does not remove the task from its current location). Where the next transformation on the genome instead turns out to be a *cluster* or does not exist, the *duplicate* transformation has no effect. Multiple instances of contiguous *duplicate* transformations have the effect of a single *duplicate*.

Where many instances of a particular task exist, it becomes necessary for subsequent *grab* transformations of that task to identify which instance of the task is being relocated. The congruency is broken by first identifying how many duplicates of the chosen task are indeed present on the VPM, obtaining the gene value immediately before the current gene and modding this value by the number of duplicates identified. This yields a value n. The nth instance

of the task encountered as the processors are visited in turn is the one that is relocated. The value obtained from the previous gene being contiguous to the *grab* ensures a very high likelihood of preserving context from one generation to the next.

Once the genome has been processed, since multiple instances of the same task on the same processor are unnecessary, the earliest instance of each task on each processor is kept while the later ones are removed in one final sweep before deadlock checking and fitness evaluation is carried out.

6. FITNESS EVALUATION

After the resulting phenotype has been checked for intra-processor deadlock and rectified where necessary, the determination of the exact start times of all nodes completes the scheduling. This is achieved by visiting the head node of each processor's execution sequence in turn, allocating the Earliest Start Time (EST) possible to the node, removing such nodes from the sequence and repeating the process. The EST value of a node later in the sequence will depend on the EST values of its predecessors, their computation times and whether a communication delay is to apply where they reside on different processors to the task in question. The EST value of the final node to be scheduled added to its computation time results in the schedule length (Parallel Time). The fitness of an individual is given by the ratio of the Serial Time divided by its Parallel Time, where the Serial Time is the time taken to complete the program on a uniprocessor i.e. the sum of the node computation times.

Where a predecessor of a given task is duplicated on a number of processors, all duplicates except the one that yields the smallest EST value for the task in question are temporarily ignored from the viewpoint of determining that particular task's EST.

7. EXPERIMENTAL SETUP

It was decided to focus on a number of standard benchmark DAGs from the literature. These graphs vary considerably in terms of their characteristics such as size, granularity and overall structure and therefore deemed to be relevant in terms of determining an initial impression of the overall performance of the system under investigation.

A standard Genetic Algorithm was applied to each problem with 50 generations per run with a total of 50 runs. A special one-point crossover operator was used whereby equivalent sections of the levels section of both parents are swapped (i.e. same cut point locations) or Processor-Task transformations are swapped with cut-points either between or within transformations. Crossover and bit-wise mutation rates were kept constant at 0.7 and 0.01 respectively. Population size was also kept constant at 400 for comparison purposes with other GA approaches in the literature, although in our case, populations only evolved for 50 generations instead of the 3000 generations used by [2]. Cluster and duplication rates varied between 0.01 and 0.05.

We compare against a variety of methods to show how general our scheme is. These include List Scheduling, Critical Path and Graph Decomposition methods as described in section 3.

8. RESULTS

The first 3 tables below compare the best Parallel Times for the different approaches for the various benchmark problems in the literature, including the performance of the system presented here (Sheahan/Ryan - SR). In all cases, a lower value for the Parallel Time implies it is a better schedule.

The first table makes a set of benchmark comparisons with some deterministic approaches across a range of diverse graphs [1]:

Graph	LC	LAST	CLANS	DSC	MCP	SR
K1	11	17	11	10	11	14
FFT1	172	146	124	124	148	146
FFT2	225	240	200	205	205	215
FFT4	710	170	405	710	710	160
SUM1	50	50	42	34	67	34
SUM2	50	55	42	34	35	34
IRR	710	840	725	605	605	680

Table 1: Comparison with Deterministic Approaches.

LC - (Critical Path) LAST - (List Scheduling) CLANS - (Graph Decomposition) DSC - (Critical Path) MCP - (Critical Path)

Table 1 shows that the system presented (SR) surpassed all other methods for one particular Fast Fourier Transform problem (FFT4), by finding a schedule (PT = 160) that completed 10 time units before its closest rival (LAST). For two other problems (SUM1 and SUM2), it matched the leading Critical Path Technique (DSC) in schedule length.

Table 2 compares the performance of the current system with a set of Critical Path techniques from the literature for two more benchmark problems, [4](A) and [5] (B) respectively. In both cases, the current system outperforms existing techniques.

Graph	Nodes	MCP	DSC	MD	DCP	SR
A	18	520	460	460	440	410
B	9	14	14	15	14	12

Table 2: Comparison with Critical Path Approaches.

MD - (Critical Path) DCP - (Critical Path)

The next table makes a set of benchmark comparisons with a GA approach [2] along with some traditional list scheduling approaches:

Graph	Nodes	ISH	DSH	CPFD	Wu	SR
P1	15	300	300	300	300	300
P2	15	500	400	400	400	420
P3	14	260	260	260	260	260
P4	14	400	310	330	330	350
P5	15	650	539	446	438	361
P6	17	41	37	37	37	36
P7	18	450	370	330	350	360
P8	16	760	760	760	760	760
P9	16	1220	1030	1040	1040	1110

Table 3: Comparisons with a GA and List Scheduling methods

ISH - (List Scheduling) DSH - (List Scheduling) CPFD - (List Scheduling) Wu - (GA)

Table 3 indicates the potential of the system to outperform the GA method, as seen in specific cases P5 and P6 above. The results also show that the system presented (SR) is capable of matching performance when compared with rival list scheduling methods (specifically problems P1, P3 and P8 above).

The final table (table 4 makes a direct comparison between the results obtained by [2] and the system presented for the same set of benchmarks. This time the *average* performances are compared with populations of 400. In this case we see that for the majority of the cases presented, the current system (SR) produced PT values that averaged out ahead of Wu over the 50 runs.

	Wu		SR	
Graph	**Best**	**Average**	**Best**	**Average**
P1	300	300	300	300
P2	400	430	420	440
P3	260	263.4	260	260
P4	330	370	350	360.4
P5	438	445.92	361	361
P6	37	37.78	36	36.86
P7	350	380.6	360	374.9
P8	760	782.8	760	775.1
P9	1040	1101.8	1110	1130.4

Table 4: Set of Average Parallel Time Comparisons

9. CONCLUSION AND FUTURE WORK

We have described a GA approach to scheduling that employs a representation which is a sequence of transformations that can be applied to a DAG that can always produce valid schedules. We have shown that by extracting information about the *mobility* of nodes in the DAG, the GA can tread a fine line between being constrained into the space of useful schedules yet still given enough freedom to explore.

We have compared this method to existing deterministic approaches on 18 different graphs and found that in the majority of cases, the method has performed remarkably well in terms of matching most of the leading methods, and in many cases has surpassed them. In making a direct comparison with another GA method, we found that while neither method clearly outperformed the other, the system presented here invariably had a lower *average* Parallel Time score, suggesting that it may be more robust. Our system also ran for 50 generations while the other GA system ran for 3000, with identical population sizes.

This paper has served to introduce our system. Future work will include more comparisons with existing techniques, scaling to larger problems and an investigation into the characteristics of the graphs that these systems find difficult.

10. REFERENCES

[1] A.A. Khan, C.L. McCreary, M.S. Jones, *A comparison of Multiprocessor Scheduling Heuristics International Conference on Parallel Processing*, Vol. 2, pp. 243-250, 1994

[2] A.S. Wu, H. Yu, S. Jin, K. Lin, G. Schiavone *An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 9, 2004

[3] Sugiyama, Kozo, Shoijro Tagawa and Mitsuhiko Toda, *Methods for Visual Understanding of Hierarchical System Structures*, *IEEE Transactions on Systems, Man and Cybernatics 11(2)*, pp. 109-125, 1981

[4] 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

[5] J.Y. Colin and P. Chretienne, *C.P.M. Scheduling with Small Communication Delays and Task Duplication Operations Research*, Vol. 39, pp. 681-684, 1991

[6] E. Coffman, *Computer and Job-Shop Scheduling Theory*, *John Wiley and Sons*, 1976

[7] T. Yang and A. Gerasoulis, *A Fast Scheduling Algorithm for DAGs on an Unbounded Number of Processors*, *5th ACM International Conference on Supercomputing*, pp. 633-642, Association of Computing Machinery, 1991