A Genetic Clustering Algorithm

Alan Sheahan & J.J. Collins & Conor Ryan
Dept. Of Computer Science and Information Systems
University Of Limerick
Ireland

Alan.Sheahan|J.J. Collins|Conor.Ryan@ul.ie

Abstract

The SCARE project incorporates the design and implementation of an automatic, software tool for generating functionally identical, parallel code from a given sequential program for execution on an asynchronous, distributed memory message passing architecture. There are two fundamental problems in automatic program parallelisation : (a) parallelism detection at the granularity level of the parallel machine (b) efficient execution of the detected parallelism (represented by a task graph) in order to balance computational loads and reduce unnecessary communication.

This paper concentrates on the scheduling stage, (b) above. It describes a feasibility study of the application of Genetic Algorithms to task clustering. Clustering is important in a scheduling sense because a parallel program can conveniently be represented as a task graph, with a node for each instruction and communication between instructions being denoted by an edge. The program can be mapped directly from a graph onto a parallel architecture, with all nodes in a particular cluster being executed on the same processor. We examine some of the popular clustering methods and describe their suitability for scheduling, before describing the application of a Genetic Algorithm to clustering. We show that not only is the GA is competitive with existing methods but an emergent ability of the GA is that of being able to discover the optimal number of processors.

1 Introduction

Until recently, parallel programming tended to be restricted to either purely academic activities or to exotic super computer systems which were normally the preserve of wealthy institutions. The advent of systems such as PVM[Geist, 1993] (Parallel Virtual Machine)/ MPI (Message Passing Interface) and Linda[Gelernter, 1985] have changed this, however, by treating a netwo rk of (possibly heterogeneous) computers as though each were a node in a parallel computer. The performance and practicality of these systems has further improved with the use of Beowulf systems, which are generally groups of Intel or Alpha-based machines on a fast (100MBit or greater) local network running a version of PVM or MPI. These systems have all the characteristics of the PVM type systems mentioned above, with the added advantage of extremely fast communication, thus allowing the possibility of increasingly fine grains of execution. Parallel processing is becoming increasingly important as more and more sophisticated techniques are being developed for areas such as simulations, engineering applications or graphics rendering. The Scare project is concerned with creating an automated tool for converting serial programs into a functionally equivalent parallel version. It consists of three stages, Program Comprehension, Autoparallelisation and Scheduling, which this paper is concerned with.

The first stage, Program Comprehension, examines a program and prepares it for parallelisation, i.e. the areas of the program most likely to benefit from parallelisation are identified, and any necessary information for the parallelisation stage is extracted. The second stage, which we call Paragen, is concerned with the actual process of parallelisation. Paragen generates a program with as much parallelism as possible, by performing a large number of transformations on the original, serial program. A description of Paragen is available in [13]. Paragen is not architecture specific, however, and generates a program that uses as much parallelism as it can extract from a program.

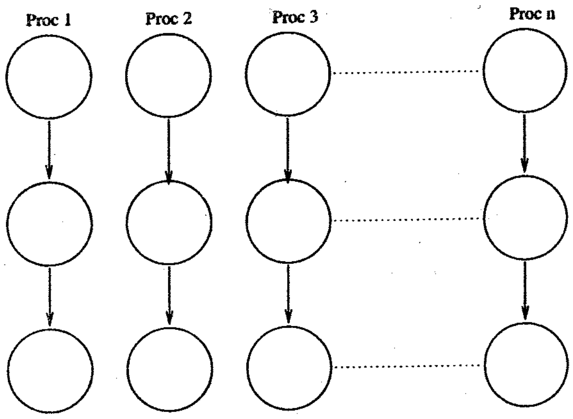


Figure 1: A highly efficient parallel program.

The ideal situation is that Paragen will generate a graph similar to that found in figure 1, in which each column can simply be mapped onto a separate processor. In this attractive, yet unlikely, scenario there would be no inter-processor communication, which would permit each processor to work independently, generating a hugely quick program.

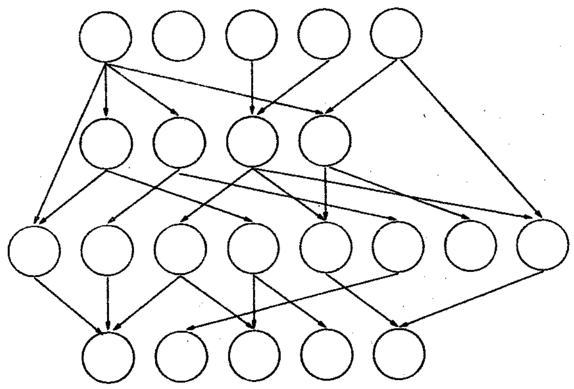


Figure 2: A more realistic looking parallel program.

Unfortunately, figure 2 is by far a more likely situation. Although Paragen extracts a large degree of parallelism, it makes no effort to remove or even reduce inter-process communication, this is because Paragen is architecture independant. What is required is a scheduler that will tune the output of Paragen to a particular architecture, reducing communication as much as possible in the process. This paper is concerned with using a GA to achieve this.

2 Multiprocessor Scheduling

The multiprocessor scheduling problem is to map a set of precedence-constrained tasks, $T_{i,(i=1:n)}$ onto a set of processors $P_{k,(k=1:p)}$ in order to satisfy a specified objective. It can be sub-divided into the following steps :

(a) Assignment of tasks to processors
(b) Ordering of tasks within each processor
(c) Determination of execution times of tasks within each processor

Mutiprocessor scheduling problems can be classified according to the amount of information available about the tasks at compile time, the extent to which they preempt and reallocate processors among jobs, the arrival pattern of jobs, application scalability (i.e. way in which an application can be expected to behave executing with a varying number of processors on a given hardware architecture) and resource requirements ..etc.

The objective from the SCARE viewpoint is to minimise the program completion time (Parallel Time) for a specified number of processors assuming a dataflow task model of execution (i.e. task preemption is prohibited). Finding optimal scheduling solutions without imposing a limitation on the number of processors and ignoring communication overhead is solvable in polynomial time [1,2]. However, taking communication into account, the problem of determining a schedule that specifically minimises the Parallel Time has been shown to be NP-hard in the strong sense [3,4]. There are a number of approaches one may adopt in attempting to address this problem :

2.1 Fully Dynamic

This type of approach is mainly used where the amount of compile-time information about tasks to be scheduled is limited and as a result all scheduling decisions are post-poned until run-time. This incurs a large run-time overhead in the form of increased execution time, necessary to resolve such on-line decisions, coupled with the restriction that it is not practical to make globally optimal scheduling decisions at run-time.

The main advantage it has over other approaches is its flexibility in the re-direction of the computational load based on runtime system load information in an effort to maximise throughput. However, the efforts endured to collect load information compete with the underlying computation during run-time, coupled with the processor reallocation cost leads to a sacrifice in both quality and speed of load balancing. In light of these issues and the advances in compiler technology facilitating the exploitation of compile-time information, it was felt that a non-dynamic approach was more appropriate.

2.2 Fully Static

Fully Static or Compile Time Scheduling relies on the availability of reasonably accurate information regarding task computation and interprocessor communication costs at compile time. The more information available, the greater the ability to construct deterministically optimal schedules. Typically, list scheduling algorithms , whereby runnable tasks are placed in a list sorted by their priorities (priorities being assigned commonly by Hu's level scheduling [8] or some variant of this) and the task with highest priority is assigned to the first available processor, are inappropriate as they ignore interprocessor communication cost (IPC) when assigning tasks to processors which is very significant at the scheduling phase.

Consequently, a 2-step scheduling approach has been chosen as a scheduling methodology suitable for the SCARE project. This involves a low complexity, Clustering heuristic (particularly suited to coarse grain parallelism applications represented by large task graphs) which is designed to minimise the communication overhead. This paper is concerned with an exploration of the performance of Genetic Algorithms when applied to the Clustering technique. Other approaches incorporate a hybridization of both dynamic and static methodologies some of which are mentioned below:

2.3 Other Scheduling Techniques

Quasi-Static Scheduling is used to handle the scheduling of non-deterministic tasks or dynamic constructs such as conditional, data-dependent iteration or recursive procedures. The most challenging problem associated with it, though, is the compile-time profiling of the dynamic constructs (involving the statistical distribution of the run-time behaviour of the constructs). This methodology is typically employed in DSP applications, where the program is executed once for every sample of an input stream. For such iterative executions, the objective is to maximise the throughput or minimise the iteration period.

Static Assignment Scheduling is where tasks are assigned to processors at compile time but the order of execution is not. A run-time scheduler, local to each processor, orders the tasks and invokes their execution based upon data availability.

The **Self-Timed** approach orders the task execution on each processor at compile time but leaves the exact firing time to be determined at run-time. At run-time, each processor waits for data to be available for the next task in its ordered list and then executes the task. Consequently, the schedule can compensate for certain fluctuations in execution times. This approach is adopted typically where there is no hardware support for scheduling (except synchronisation primitives) or where data dependence is not a crucial issue. It is applied usually in the areas of scientific computations and Digital Signal Processing (DSP).

The choice between all types depends on compromises between hardware cost, performance, flexibility and the amount of data-dependent behaviour in the expected application.

3 Task Clustering

The Clustering technique itself requires that a data dependence graph (DDG) representation of the parallelism be extracted initially from the data dependency analysis stage. The nodes of this graph are then mapped onto labelled tasks, which are defined as indivisible units of computation. The resultant partitioning, containing the set of task nodes and communication/dependency edges, may be represented by a Directed Acyclic Graph (DAG). Now, the mapping of the nodes onto an unbounded number of completely connected, virtual processors (clusters) is carried out in order to minimise the Parallel Time. This pre-processing step is known as Clustering (also known as internalisation pre-pass or processor assignment) completing the first stage of the process.

The second stage merges the clusters into p completely connected, virtual processors corresponding to p physical processors in the parallel architecture, using a load balancing algorithm. Next, the mapping of the virtual processors onto the physical ones is done so as to minimise the total communication cost between physical processors, taking processor distances in the parallel architecture into account. Finally, the ordering of task execution within each processor is determined so as to minimise the Parallel Time.

3.1 Dominant Sequence Clustering

All Clustering algorithms in the literature attempt to find the trade-off point between parallelism exploitation and communication minimisation. Heuristic algorithms such as Kim & Browne [5], Sarkar[3] and Wu & Gajski [6], proposed in the literature have high computational complexity (due to a search for the longest path of the graph at each step of the algorithm) and are not optimal for primitive DAGs (due to the zeroing of edges that do not explicitly reduce the Parallel Time).

The Clustering algorithm that we have chosen for our feasibility study is that proposed by Gerasoulis, Venugopal and Yang [7] called Dominant Sequence Clustering. This has been shown to perform extremely well in terms of producing a schedule for large DAGs with near-optimal minimisation of Parallel Time (PT). Its main advantage over the other clustering algorithms is that it clusters with low computational complexity (O(v+e)logv time complexity). Also, it produces optimal schedules for primitive DAGs such as fork, join or coarse grain tree graphs.

It operates by incrementally identifying the critical path of the scheduled DAG known as the Dominant Sequence (DS). A scheduled DAG defines not only the partitioning of the tasks onto processors but the local execution order aswell. Since the length of the Dominant Sequence is equivalent to the Parallel Time, the algorithm attempts to reduce the length of the DS thus reducing the PT. The DS is minimised using an edge-zeroing strategy, such that two or more nodes connected by zeroed edges execute on the same processor.

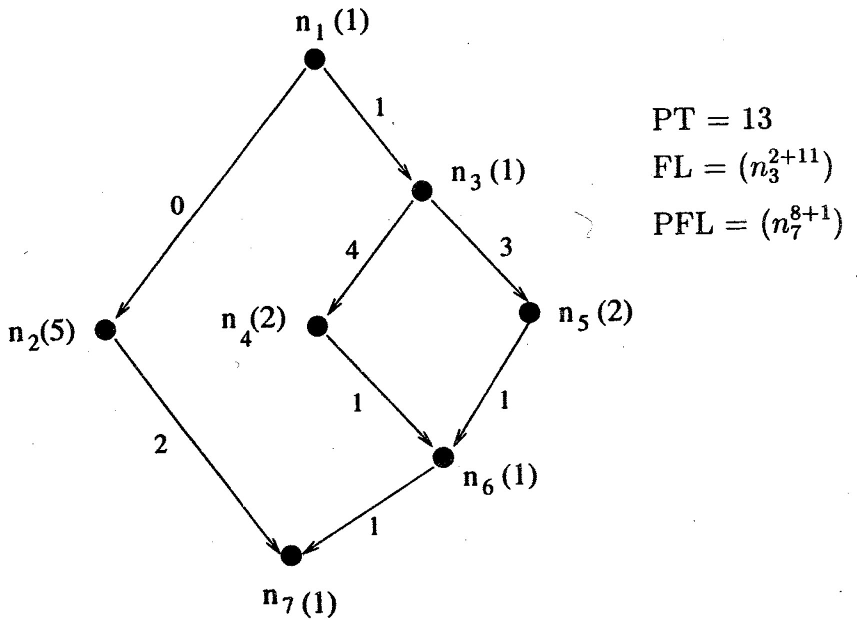


Figure 3: Dominant Sequence Clustering

Two prioritised lists are updated as the algorithm progresses, a free list, FL, containing a list of task nodes whose predecessors have already been scheduled and a partially-free list, PFL, containing a list of task nodes which has at least one, but not all, scheduled predecessors. Prioritisation of a node comprises the sum of its maximum indegree edge from a scheduled predecessor (known as the startbound of the node) and the longest exit path from the node to a sink (known as the node level). This is denoted by the superscript associated with each node in the FL and PFL in Figure 3.

In this example, n_1 and n_2 have already been scheduled in that order on the same processor resulting in the freeing of n_3 and the partial-freeing of n_7. The Dominant Sequence path is n_1, n_3, n_4, n_6, n_7 yielding a Parallel Time of 13. At any stage of the algorithm, the idea is to reduce the startbound of a node in the DS (by zeroing one of its incoming edges) thus reducing the PT. The DS path, itself, must be going through either the head of the FL or the PFL (or both). Nodes are always selected from the head of the FL for scheduling (since the partially free nodes are not ready to be scheduled) and zeroing is only carried out if the startbound of the node is seen not to increase. In such a case the target node is scheduled after the previously scheduled node in the same cluster (for execution on the same processor). Otherwise, the target node is scheduled for execution on a new processor. In either case, scheduling of the node incorporates the designation of a definitive execution time and deletion of the node from the free list. In Figure 3, the target node, n_3, is scheduled on a different processor as its startbound will increase from 2 to 6 if scheduled on the same processor as n_1 and n_2.

At some stage it may happen that the DS goes through the head of the PFL (in which case the priority of the head of the PFL is greater than the priority of the head of the FL). We obviously cannot schedule this node as it is only partially free and scheduling the head of the FL may pose a problem if we are tempted to reduce it's startbound, as this could possibly affect the potential reduction of the startbound of the PFL head node and hence affect the Parallel Time minimisation potential. Hence, we schedule the head of the FL and only reduce its startbound if it is not seen to affect the reducibility of the PFL head.

4 Genetic Algorithms

Genetic algorithms (GAs) are probabilistic search methods based on the principles of neo-Darwinian evolution. Two pillars of this model are the mechanics of natural selection and survival of the fittest [9]. GAs are traditionally used as optimisation tools, but now, there is a greater emphasis on the role of GAs within the context of genetic learning and artificial evolution [11]. The focus of design in GAs is on the representation and fitness, and not the processing mechanism itself which has a widely accepted qualitative formulation [9]. As this suggests, the GA paradigm referred to is portable from domain to domain, without requiring prior knowledge about the structure of the domain topology and any implicit correlations between the problem and representation.

A GA consists of a population of solution strings or chromosomes. The string itself is known as the genotype. The phenotype is the mapping of the genotype to the environment or problem domain. Fitness is a measure of the viability of the phenotype within the context of a particular environment. In GAs, an initial population of strings is generated randomly. The process of evolution is applied in which parents of above average fitness in the population are selected for reproduction, and their offspring added to the next generation of individuals. GAs use probabilistic transition rules as opposed to deterministic ones. Inherent to GAs is the concept of implicit parallelism. The genotype is composed of variants of different schemata or building blocks. Despite the fact that GAs process only n individuals per generation, n^3 schemata are actually processed, where n is the number of parents selected for reproduction. Critical to GAs is the specification of a fitness function which is the designer's most important task when implementing a GA. The fitness function specifies constraints that the GA will attempt to satisfy.

It is worth noting that processor scheduling using GAs is a one pass computational paradigm whereas deterministic methods employ two passes. The first pass employs methods such as Dominant Sequence Clustering (DSC) which map the DAG onto a multi-processor machine with infinite processing units. The second pass maps the output of pass one onto an architecture with a specified number of processors. In contrast, for genetic scheduling, the number of processors that the DAG is mapped onto is specified initially.

4.1 Representation

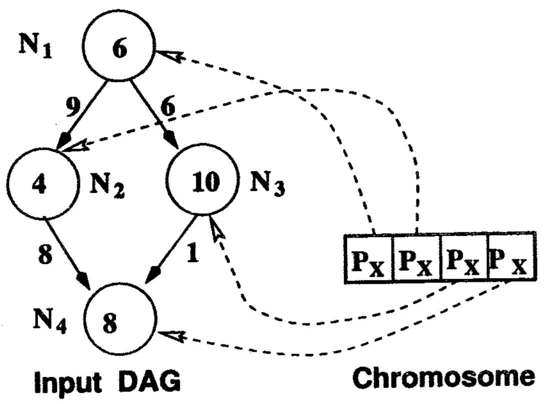


Figure 4: Left: an example of an input DAG which is sorted topologically, each node having a computation time and weights representing communication costs. Right: mapping of chromosome which represents processors to nodes.

The GA uses a fixed length Gray encoded string to represent processor scheduling. Each string consists of genes, the number being identical to the number of nodes in the input directed acyclic graph (DAG), which is sorted topologically. The problem of deriving undesirable correlation between representation and the problem domain is overcome by the use of a Gray code which preserves numerical adjacency in Hamming space [12]. This has the property that any two points next to each other in the problem space differ by only one bit, thereby reducing the possibility of obliteration of fit schemata or building blocks under the effects of mutation. Another advantage of using this scheme over a floating point scheme is that it is unnecessary to explicitly define a mutation operator, thus eliminating the introduction of hidden biases.

4.2 Fitness Function

Critical to the evolution of better solutions over time in GAs, is the evaluation of fitness which is a measure of the success of the genotype to phenotype mapping for a specific problem doamin. Fitness is a measure of the performance of a string, and in this case is a metric of the speedup gained from mapping of the DAG to a multi-processor architecture, compared to the time required for a serial execution on a single node machine. The serial execution time is calculated by summing the computation time of all nodes in the graph. Using serial to normalise the fitness of the string yields an online performance measure. AA string with a fitness less than 1 yields poorer performance than a single node machine, fitness of one indicates equivalent performance and greater than 1 indicates the percentage gain in speedup.

Processors in the string are mapped to the corresponding nodes in the DAG. Fitness is calculated using critical path analysis, in which the DAG is viewed as an activity node graph, in which the edges represent precedence relationships. For each node n and given

- Computation time t_n.
- m input edges c representing communication costs from its m predecessor nodes.
- Allocation of node n to processor p_n.

its earliest completion time EC_n is calculated as follows:

$$\begin{aligned} EC_1 &= t_1 \\ EC_n &= \max_{i=1}^{m}\Big(EC_i + p_n + \alpha_{in}\Big) \\ &\quad \text{where} \quad \alpha_{in} = \begin{cases} 0 & \text{iff i and n on same processor} \\ c_{in} & \end{cases} \end{aligned} \tag{1}$$

The fitness F of string s which is mapped onto a DAG of d nodes is:

$$F_s = \max_{i=1}^{d}\Big(EC_i\Big) \tag{2}$$

For all pairs of independent nodes on the same processor ie no edge from one to the other, a transitive edge of weight 0 was inserted into the adjacency matrix resulting in a scheduled DAG. However, applying this rule in real world applications will lead to a combinatorial explosion in the search space. We are evaluating the heuristic that only transitive edges need be inserted for topologically adjacent independent nodes on the same processor. We are currently working on using Dijkstra's shortest path algorithm to compute earliest completion time. The running time for this is $O(\mid E \mid + \mid V^2 \mid)$.

4.3 Operators

During reproduction in which two parents are selected and their genetic material combined to form two children, three genetic operators with associated probability factors are applied.

- Crossover: this operator randomly partitions the parent strings and recombines partitions in the genotype of their children. Several crossover mechanisms are available, one point, two point and uniform. This is an exploitation operator.
- Mutation: this operator changes the value of a randomly selected bit in the string. Because a Gray encoding scheme uses only two symbols, 0 and 1, mutation just involves flipping the selected bit. Mutation helps to maintain genetic diversity in the population of strings and thus facilitates exploration of the search space.
- Duplication: a randomly selected gene in the child string is replaced by another randomly chosen gene from the same string.

5 Empirical Results

A population size of 50, run for 500 generation equivalent cycles.1-point crossover was used with a probability of 0.5. Mutation probability was set to 0.001 and 0.5 for duplication. A steady state replacement policy was used.

Genetic Clustering			
	Problem Domain		
	1	2	3
No. nodes in DAG	7	12	12
No. Processors	2	6	2
Serial Time	13	30	19
Theoretically Max DSC PT	9	17	12
Maximum GA PT	9	17	13
Achieved by generation	2	60	55
Mean processing time per generation	10.38ms	24.05ms	22.2ms

Table 1: Results for genetic clustering. PT is parallel time. The maximum GA PT for each problem domain was achieved in all test runs

The viability of genetic clustering was evaluated on three problem domains. The first DAG as depicted in fig. 5 is a simple, yet classic one consisting of 7 nodes. The next two DAGs tested are larger, the second as shown in fig refprob3 possesses a finer measure of granularity resulting in the general increase in communication costs throughout the DAG. For each problem, 20 tests were run and average fitness and generation times calculated. The results are depicted in table 1. It is interesting to note that while problem 2 specified that six processors were available, GA clustering scheduled the tasks using only five processors in all test runs, while still achieving theoretically maximum parallel time (PT). This is surprising because the fitness function does not incorporate any metric on processor usage or load balancing.

The results in figs. 5 6 and 7 illustrate and contrast both deterministic clustering using DSC with load balancing, against genetic processor scheduling. For problem domain 1 in . refprob1 both DSC and the GA yielded identical clusters at all times. The results demonstrate that genetic clustering can yield comparable performance to deterministic methods. For problem domains 1 and 2, the GA was able to optimize the processor schedule within a very short time span. Our research is now directed at implementation of deterministic methods to enable benchmarking of the GA method. Of particular interest will be the issues of scalability and load balancing.

6 Conclusions

We have shown that a GA is competitive with existing methods at capable of generating graph clusters. The results in table 1 illustrate how similar the results are. However, the GA also appears to demonstrate the ability to determine the optimal number of processors, while existing methods rely on the user to supply this information to the system.

References

[1] R.L. Graham, "Bounds on Multiprocessing Timing Anomalies", *SIAM J. Appl. Math.*, Vol 17,pp. 416-429, 1969.

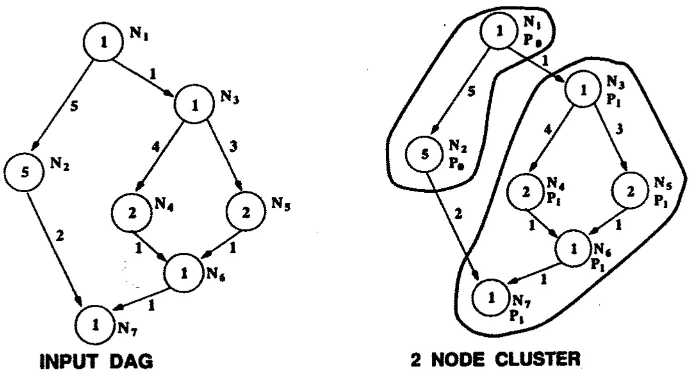


Figure 5: Shown for problem 1 and a 2 node machine: input DAG (left), resultant clustering which was identical for DSC and the GA (right).

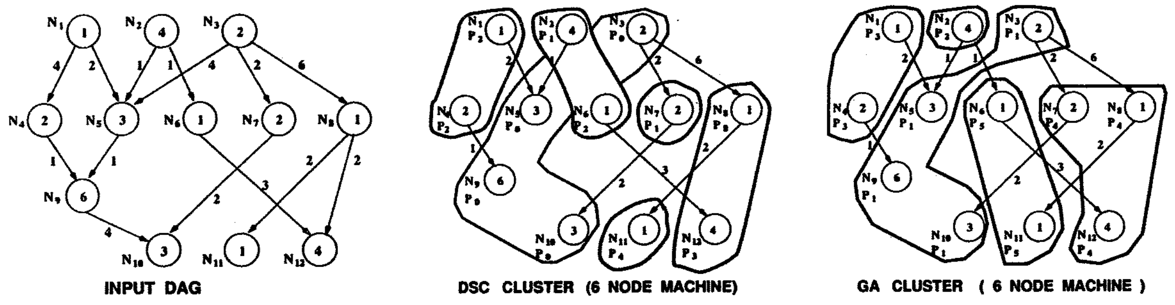


Figure 6: Shown for problem 2 and a 6 node machine: input DAG (left), resultant clustering using DSC (middle), and and arbitrarly chosen cluster using the GA (right). Notice that the GA only uses five of the six processors.

[2] J.K. Lenstra and A.H.G. Rinnooy Kan, "Complexity of Scheduling under Precedence Constraints", Operation Research, Vol 26:1,1978.

[3] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors'." *MIT Press*, 1989.

[4] Ph. Chretienne, "Task Scheduling over Distributed Memory Machines", *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, North Holland, 1989.

[5] S.J. Kim and J.C. Browne, "A General Approach To Mapping Of Parallel Computation Upon Multiprocessor Architectures" *International Conference ON Parallel Processing*, Vol 3,pp. 1-8,1988.

[6] Min-You Wu and D. Gajski. "A Programming Aid for Hypercube Architectures" *The Journal of Supercomputing*, Vol 2,pp. 349-372,1988.

[Geist, 1993] Geist, G (1993), "PVM 3 Beyond Network Computing," in *Lecture Notes in Computer Science 734, 2nd Int. Conf. of the Austrian Center for Parallel Computation* pp. 194-203, Gmunden, Austria : Springer Verlag.

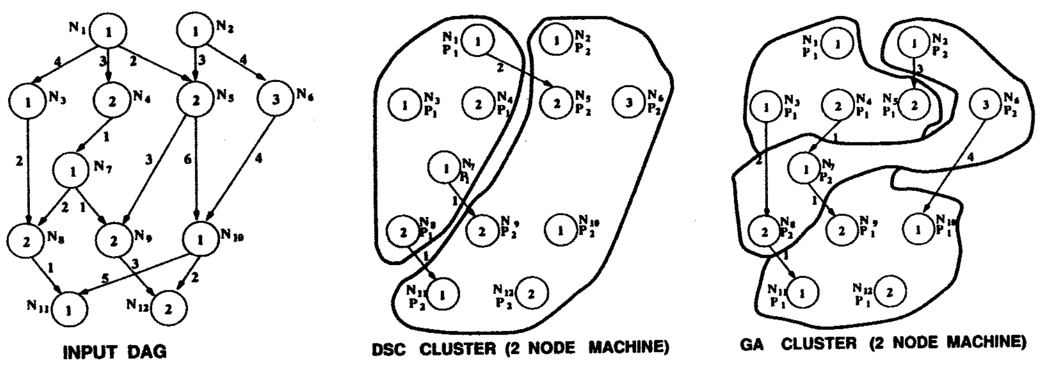


Figure 7: Shown for problem 3 and a 2 node machine: input DAG (top), resultant clustering using DSC (left), and and arbitrarly chosen cluster using the GA (right).

[Gelernter, 1985] Gelerneter, D. (1985), "Parallel Programming in Linda", Technical Report 359, Yale University Department of Computer Science.

[7] A. Gerasoulis, S. Venugopal and T. Yang, "Clustering Task Graphs for Message Passing Architectures" *Proceedings of ACM International Conference on Supercomputing*, Amsterdam,pp. 447-456,1990.

[8] T.C. Hu, "Parallel Sequencing and Assembly Line Problems" Operations Research, 9(6),pp. 841-848,1961.

[9] Goldberg, D. (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Publishing.

[10] Horn, J. and Goldberg, D. (1994). Genetic Algorithms Difficulty and the Moadality of Fitness Landscape. In *Proc. of the Foundations of Genetic Algorithms 3 (FOGA3)*, pages 243–270.

[11] Sipper, M., Sanchez, E., Mange, D., Tomassini, M, Pérez-Uribe, A. and Stauffer, A. (1997). A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. *IEEE Trans. on Evolutionary Computation*, Vol. 1, No. 1, pages 83–97.

[12] Whitley, D., Mathias, K., Rana, S. and Dzubera, J. (1996) Evaluating evolutionary algorithms. Artificial Intelligence, 85: pages 2745–2761.

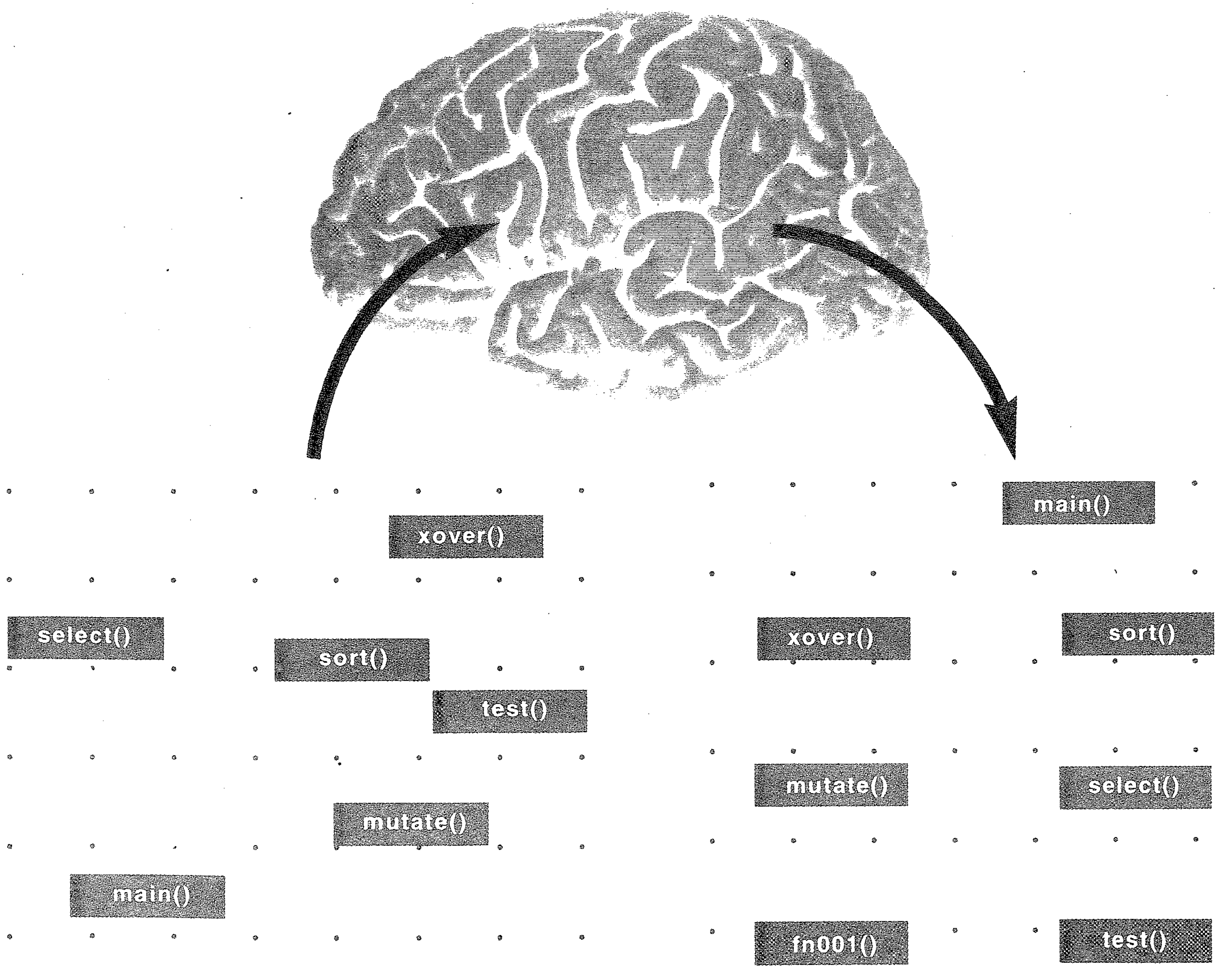
[13] Ryan, C. and Ivan, L. (1999) Evolving Equivalent Parallel Programs : Sequences and Iterative Instructions *Proceedings of Scase99.*

[illegible]

SCASE'99

SOFT COMPUTING APPLIED TO SOFTWARE ENGINEERING

UNIVERSITY OF LIMERICK, IRELAND
APRIL 1999



edited by Conor Ryan and Jim Buckl

SCASE'99

Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering

Limerick, Ireland.
12th-14th April, 1999

Editors:

C. Ryan and J. Buckley

ISBN: 1-874653-52-6
Limerick University Press.

SCARE
Soft Computing and Re-Engineering Group.
University of Limerick, Ireland.
Funding by Forbairt.