8-1995

A Comparison of Methods to Allocate Tasks on a Multiprocessor System

Kelly Cousineau

Recommended Citation

Cousineau, Kelly, "A Comparison of Methods to Allocate Tasks on a Multiprocessor System" (1995). *Master's Theses*. 4245.
https://scholarworks.wmich.edu/masters_theses/4245

A COMPARISON OF METHODS TO ALLOCATE TASKS ON A MULTIPROCESSOR SYSTEM

by

Kelly Cousineau

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
August 1995

ACKNOWLEDGMENTS

Without a doubt, the most important one in my life and hence, in the completion of this project, is Jesus Christ, my Lord and Savior. It is because of Him only that I've been able to pursue and accomplish this goal. Thank you Jesus.

It is here also that I admit Darwin has created an interesting problem solving method as discussed herein. Nonetheless, I must emphatically proclaim that the one and only God, the trinity (Father, Son and Holy Spirit) created me, every other human being and every creature that walks this earth.

My husband Bill has been extremely loving, patient and encouraging. Through our entire courting relationship and first year of marriage he has made significant sacrifices to ensure I finish this effort. Thank you and I love you.

To the professors who have kept me moving, Dr. Gupta, Dr. Greenwood and Dr. Trenary, thank you for your patience through the lull times, thanks for your great ideas, and thanks for all the things I've learned.

Kelly Cousineau

A COMPARISON OF METHODS TO ALLOCATE TASKS ON A MULTIPROCESSOR SYSTEM

Kelly Cousineau, M.S.

Western Michigan University, 1995

The task scheduling problem is defined as a sequential algorithm, with individual tasks, t_1, t_2, t_3, ..., having associated execution times to be ported to a multiprocessor system. To determine the fastest parallel implementation is known to be an NP-complete problem, therefore heuristic techniques are employed to arrive at the best solution.

Some of the traditional methods used to solve this problem, are the list scheduling algorithm and the simulated annealing algorithm. However, evolutionary techniques are now also a consideration with the increased computation power available.

Evolutionary techniques, based on the popular theory of evolution, consist primarily of a solution represented in a chromosome data structure. Maintaining a population of chromosomes, generational transitions are accomplished through reproduction of the best solutions. The objective is to improve solutions by making small random changes and repeating the process.

The results found in this effort confirm the viability of the technique. Another observance was the evolutionary technique, as implemented here, had more overhead and therefore did not perform as exceptionally as the simulated annealing method.

TABLE OF CONTENTS

Table of Contents--Continued

LIST OF TABLES

LIST OF FIGURES

List of Figures--Continued

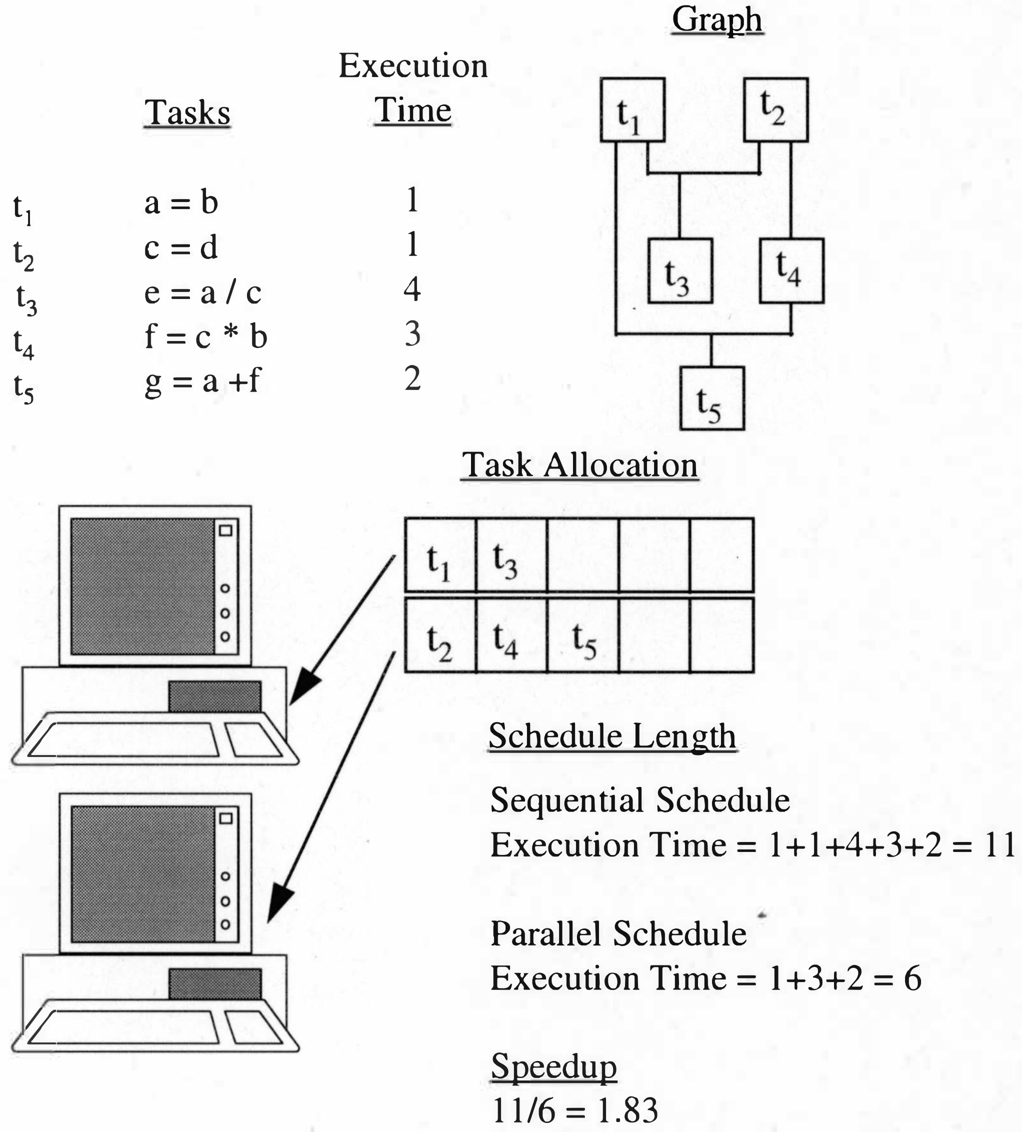
INTRODUCTION

The Problem

As computing power has increased, so has the complexity of the problems we are able to solve. Some problems have become so complex, polynomial time solutions are not believed to be possible. In these cases, heuristic techniques must be sought to estimate solutions close to optimal.

While increased computing power may have increased problem complexities, it has also provided generous resources for solving these problems. One such resource is parallel processing. The limit imposed by the typical uni-processor machines (processor can perform one operation at a time) is being removed with the advent of parallel machines (or multiprocessors) which are able to perform more than one operation simultaneously. As we begin developing parallelizable algorithms, however, we realize the need for assigning tasks to different processors to capitalize on the multi-computing power available. This is the Task Scheduling problem. A broad description of all variations of task scheduling is presented in [4].

Figure 1 presents an example of the task scheduling problem. Given a sequential algorithm, individual tasks, t_1, t_2, t_3, ..., have associated execution times (the computation time necessary to perform that particular task). In certain instances, tasks may not be performed until other specific tasks have been completed (precedence constraints). A graph may be constructed to illustrate the flow of tasks. In the example,



	Tasks	Execution Time
t_1	a = b	1
t_2	c = d	1
t_3	e = a / c	4
t_4	f = c * b	3
t_5	g = a +f	2

Schedule Length

Sequential Schedule
Execution Time = 1+1+4+3+2 = 11

Parallel Schedule
Execution Time = 1+3+2 = 6

Speedup
11/6 = 1.83

Figure 1. An Example Depicting the Problem of Task Scheduling.

the graph depicts the need to execute task t_2 before task t_4, and t_4 before task t_5. Based on those precedence constraints, tasks are then allocated or assigned to the multiple processors. Finally, a schedule may be computed based on the unique allocation of tasks. Again referring to the example, the parallel schedule length is based on tasks t_2, t_4 and t_5 since t_1 can be performed at the same time (and has the same execution time) as t_2, and while t_3 takes four time units, there are no tasks waiting for it to complete. The parallel execution of tasks t_4 and t_5 require two and three time units, respectively, and therefore, the total time of six units is calculated.

Notice that allocating tasks which must be scheduled within precedence constraints, that may be assigned to any of several processors, resulting in the fastest possible schedule, is an NP-complete problem. [7]

Importance of the Problem

Given the nature of the task allocation problem, the impending availability of multiprocessing machines and the general pace of business in the world today, it is no wonder that solutions must be produced fast! The largest advantage of parallel processing is speed. In this context, speed is measured based on the time between starting and ending a process. Those wanting the new parallel processing technology desire to have their computing applications produce results in a fraction of the time they are currently realizing with uni-processors. Of what use will this improvement be if a haphazard multiprocessor allocation scheme results in only a modest increase in speed? This illustrates the need to determine the best (or estimated best) allocation scheme for an application very quickly. To this end, a variety of methods are investigated herein.

Solutions for the task scheduling problem developed thus far include the list scheduling greedy algorithm developed by R. L. Graham [10]. Simulated annealing has been used to create schedules for use in the simulation of avionics systems [3]. A few variations of genetic algorithms have also been successful as discussed in [2].

Goals

The scope of this work is limited to static scheduling of tasks wherein a solution is derived before the actual running of the application. The availability of parallel processing has advanced the use of evolutionary techniques for solving this problem. In this paper, several of these methods are applied to assigning the tasks of Burg's Algorithm to two processors. The goal herein is to show that evolutionary algorithms provide viable solutions when applied to the task allocation/scheduling problem. Popular techniques for task scheduling problems such as list scheduling and simulated annealing were chosen to compare the evolutionary algorithms against. The evolutionary algorithms, list scheduling and simulated annealing methods are fully defined in the next section, Background and Related Work. Burg's algorithm is described in detail in the section entitled Techniques Used in this Effort.

Observations - Statement of Results

The two measures used in this comparison were speedup and cost. Speedup is the improvement that parallel processing provides over sequential processing. Referring back to Figure 1, the speedup in the example was computed as the schedule length on one processor divided by the schedule length when the tasks have been implemented on two processors. The significant cost incurred in task scheduling is the time required to arrive at the schedule. Therefore, quoted results include the number of minutes required to find the speedup listed. Table 1 summarizes the median results of one hundred runs of each of the methods evaluated. The list scheduling method performed as expected, not an outstanding speedup, but arrived at very quickly. The genetic algorithm converged in about a half an hour to that same speedup. The evolutionary strategy searched longer (three quarters of an hour) and found slightly better speedups. As shown in the Conclusions section later, the trend in both the genetic algorithm and evolutionary strategy is that the longer time spent in the search, the better the final speedup obtained. The simulated annealing method, however, was able to find obviously high speedup values in only an hour. This cause of this phenomenon is believed to be the overhead necessary to carry a population in the

Table 1

Summary of Results of Methods Evaluated

Method	Time (min.)	Speedup
Genetic Algorithm	32	1.47
Evolutionary Strategy	43	1.48
List Scheduling	16	1.47
Simulated Annealing	58	1.52

evolutionary techniques versus maintaining only one solution in the simulated annealing method.

BACKGROUND AND RELATED WORK

Evolutionary Algorithms: Basic Concepts

The basic methodology involved in evolutionary algorithms is based on the popular defination of evolution (often attributed to Darwin). Simply stated, in any particular generation, individuals undergo replication and the resulting offspring contain some combination of traits from the parent(s). Initially, individuals are randomly created and then proceed through reproduction. Any individuals found acceptable repeat the same cycle in the next generation. Applying this theory to problem solving minimally involves:

1. Represent solutions to the problem in a manner that is inclusive, accurate and enables easy implementation and manipulation (the chromosome/individuals).
2. Determine an objective method to evaluate and compare the goodness of chromosomes (the fitness).
3. Develop a decision criteria to select chromosomes for reproduction and continued existence (selection criteria).
4. Establish meaningful operators to create new chromosomes based on slight perturbations within one or between several chromosomes (the operators).
5. Analyze probabilities for chromosome reproduction operators and total number of chromosomes within a generation (the variables).

6. Devise a measurement scheme to indicate the end of reproduction cycles and report the best solution (stopping).

These elements are now described in detail and briefly illustrated in Figure 2.

The Chromosome

Ideally, the chromosome should encode problem parameters in their entirety. Chromosomes lacking a particular detail of the solution cannot be compared objectively. Further, the data structure chosen to represent the solution must be easily manipulatable and enable quick evaluation. The benefits of reviewing an entire population of chromosomes in every generation dwindle if computing the fitness of each is too cumbersome.

Binary string representations are often chosen for quick evaluation and ease of operator manipulation. However, non-binary string solutions frequently provide a more inclusive representation. Additionally, if binary solutions must be encoded and decoded, the savings of evaluating and manipulating are offset. In either case, recognition of the solution elements is beneficial for intermediate reporting and comparisons during the developmental stages.

Fitness

Since chromosomes are representing solutions to a problem, some objective means must exist to compare the ‘goodness’ of different chromosomes. If comparing solutions having two parameters (such as number of processors required and time to compute), a fitness function comprised of

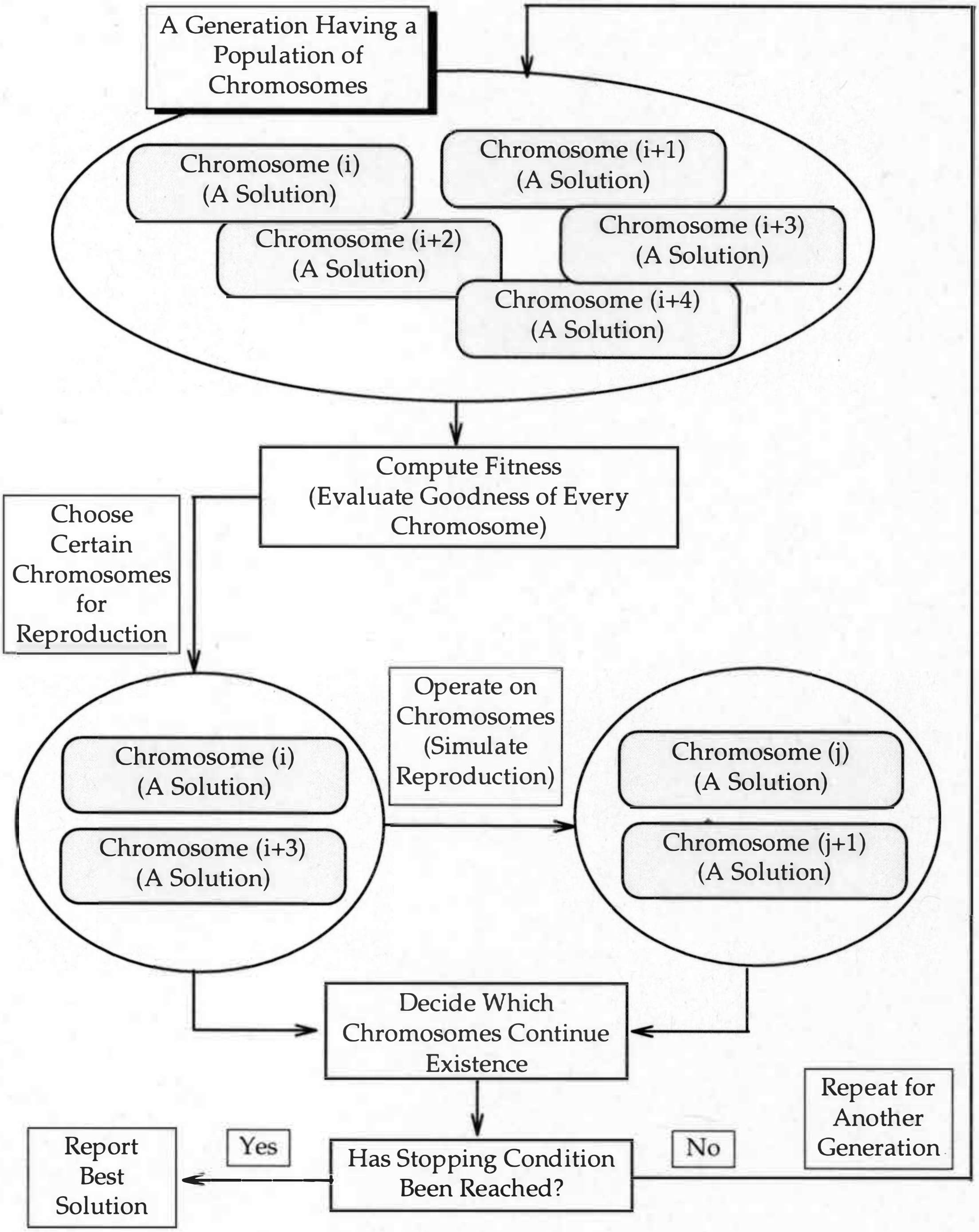

A Generation Having a Population of Chromosomes
Chromosome (i) (A Solution)
Chromosome (i+1) (A Solution)
Chromosome (i+2) (A Solution)
Chromosome (i+3) (A Solution)
Chromosome (i+4) (A Solution)
Compute Fitness (Evaluate Goodness of Every Chromosome)
Choose Certain Chromosomes for Reproduction
Chromosome (i) (A Solution)
Chromosome (i+3) (A Solution)
Operate on Chromosomes (Simulate Reproduction)
Chromosome (j) (A Solution)
Chromosome (j+1) (A Solution)
Decide Which Chromosomes Continue Existence
Has Stopping Condition Been Reached?
Yes
Report Best Solution
No
Repeat for Another Generation

Figure 2. The Elements and Flow of Evolutionary Algorithms as a Problem Solving Method.

both parameters would be necessary to objectively evaluate chromosomes. In the case of schedules, often the fastest schedule is the best.

The Operators

The operators play a crucial role in creating offspring from one or more parent chromosomes. The following operators are typical in most evolutionary algorithms.

Inter-Chromosome - Recombination

Relating again to humans, when two parents reproduce, they create a child having a combination of both parents' genes. In evolutionary algorithms this concept is generally accomplished by swapping a small quantity of genes (details of the solution contained within a chromosome) between two chromosomes as shown in Figure 3.

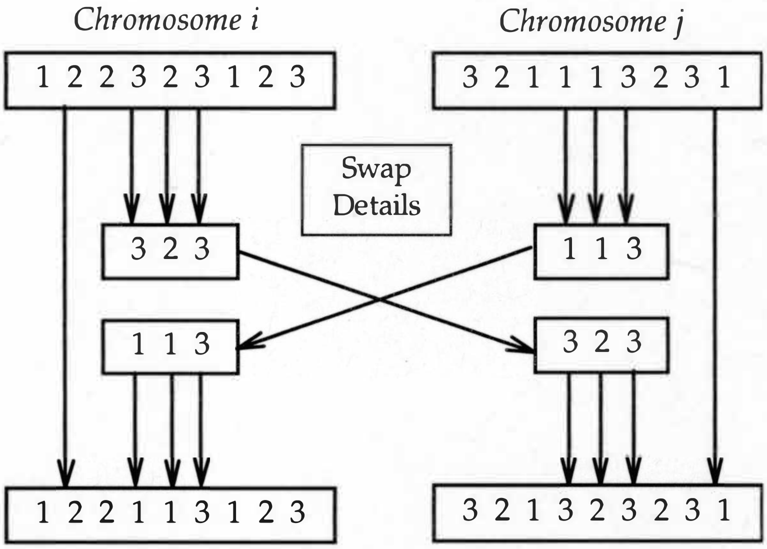


Figure 3. Inter-Chromosome: Recombination of Two Chromosomes to Create Two New Chromosomes.

Intra-Chromosome - Mutation

In humans, mutation is often the cause of birth defects. Evolutionary algorithms are inherently random and consequently the manipulation of genes within a chromosome has a chance of resulting in a better solution in addition to the chance of creating a worse solution. Figure 4 depicts the mutation of a gene within a chromosome solution.

The Variables

The major contributing elements need to have numeric values determined and assigned such as: how many chromosomes should exist in a given population? Should all chromosomes undergo mutation and recombination? Discussion of values for these variables is given below.

Population Size

As in any optimization problem, there is a tradeoff. Here, the more chromosomes there are in a population, the more solution space that can

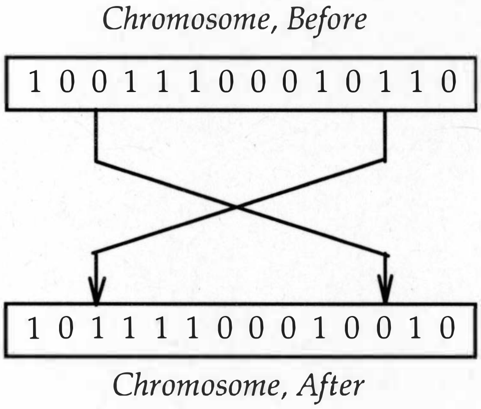


Figure 4. Intra-Chromosome: Mutation of a Gene Within a Chromosome.

be explored in one generation. Nevertheless, computers are still limited in speed and memory, and the larger the population in any one generation, the longer it will take to evaluate and move on.

Traditionally, population size has been a variable that is fixed at the start of the program run. Later in this paper, it is investigated as a dynamic variable (also called self-adapting behavior) where the size of the population reduces or increases based on the goodness of the current generation and best chromosomes to date. A more detailed discussion of the purpose and benefits will be provided in the Variable Population part of this section.

Mutation and Recombination

The common operators discussed previously could be applied to every chromosome in every generation although this would tend to create an extremely diverse population increasing the amount of wandering around before converging on a good solution. The probabilities typically used are based on what has been observed in the human equivalents, namely, probability of mutation is very small (10 - 30%). The probability of recombination generally rides above 50%. Typically, probability of mutation and probability of recombination add up to one hundred percent. Implementing the percentages is easily accomplished as shown in Figure 5.

Additional values related to the mutation and recombination operators include determining the genes to be swapped. In mutation, one gene may be moved or two genes may be swapped. In either case, the location of the gene(s) must be selected. Often this is randomly chosen, possibly within boundaries. Similarly, in recombination, the number of genes to be swapped, N, from position A in chromosome 1 and position B in chromosome 2, where N, position A and position B must all be selected. In some cases, N genes may move from one chromosome to the other without the mutual swapping.

These variables should be analyzed in detail based on the types of solutions involved. It may be that in a specific solution representation, certain mutations could cause chromosomes to become infeasable. This would lead to establishing the boundaries mentioned above.

```
for chromosome = 1 to Population_Size do

    (* 0 <= random_number <= 100% *)

    if  random_number < Probability_Mutation
          then mutate chromosome

    else
    if  random_number < Probability_Recombination
          then recombine chromosome

end  (* for population size *)
```

Figure 5. Implementing Probability of Mutation and Recombination.

Stopping

Another major factor that will affect how long the process runs and how good the 'best' solution produced is the "stopping" criterion. There are obviously many choices, though some have more merit than others. A very simple method would be computation time, say 25 minutes. Another simple method is a pre-defined number of generations, but how many?

Another simple strategy is to wait for the solution to reach a particular value, (possibly a percent improvement over initial values) but in the best cases, this could cause premature ending and put a bound on the result. One of the most logical stopping criteria is to determine a value for convergence, X, where the process halts when there has been no improvement in the result for X generations.

Selection Criteria and Self Adapting Behavior Are Specific to Method

The flow of the evolutionary algorithms is structured around another aspect of Darwin's philosophy, the fittest individuals survive. Applying this to problem solving: if a solution is good, it will most likely reproduce good solutions. Therefore, good chromosomes should be kept around for future generations.

The decision of which chromosomes go forward to subsequent generations is handled differently in the distinct algorithms. Briefly, the genetic algorithm uses a stochastic approach [9] while the evolutionary strategy applies a deterministic method [23]. The variable population technique [1] replaces the selection decision with a life strategy thus allowing an age value to systematically remove chromosomes in the appropriate generation.

Self adapting behavior is at the forefront of this research today. Instead of running countless experiments to arrive at the best input variables for the process, why not let the algorithms modify these values based on the solutions under evaluation? The variable population algorithm has implemented this approach such that the number of chromosomes in a generation is based on the current chromosomes' fitness. Additional self adapting behaviors are discussed later in this work and many more are currently under investigation.

The variations presented here will be discussed in greater detail along with the pioneering efforts in the algorithms' respective sections coming up next.

The Genetic Algorithm

Most of today's work in the genetic algorithm field is based on Holland's work [15] which has been around for over twenty years. In that work, Holland used simple binary encodings and developed the fundamentals of evolutionary programming. Some early distinctions of the genetic algorithm were the use of the cross over (or recombination) operator and the binary encoded chromosomes.

The selection of 'best' individuals has been primarily a stochastic approach. Here, chromosomes are given a weighted chance of selection where the weighting is based on their respective fitness relative to others in the same population. This allows those with relatively high fitness a much greater chance of selection, and thereby survival into subsequent generations. However, even those chromosomes with poor fitness values still have a chance, although the weighting is very low. Also, to maintain the same population size from generation to generation, selections are made until the population is complete. This tends to enable selection of the better chromosomes more than once, and similarly, the lesser chromosomes, probably not often, if at all.

This selection criteria typically leads to a quicker convergence to a good solution. The down side is that the solution could possibly be a local optimum instead of the global optimum. Observing Figure 6, notice several sub-optimal peaks in the solution landscape. These solutions are obviously better than many of the choices, but not the best. When 'good' chromosomes are consistently selected (with greater probability than bad chromosomes) for reproduction, there is a tendency to climb the hill to the nearest optimum. The 'worse' solutions, if kept, sometimes will move the search away from a local optimum allowing it to find the global optimum.

A final characteristic tendency with the genetic algorithm is the order of selection compared to operation. In general, selection of the 'parents' is made before operating to create offspring chromosomes.

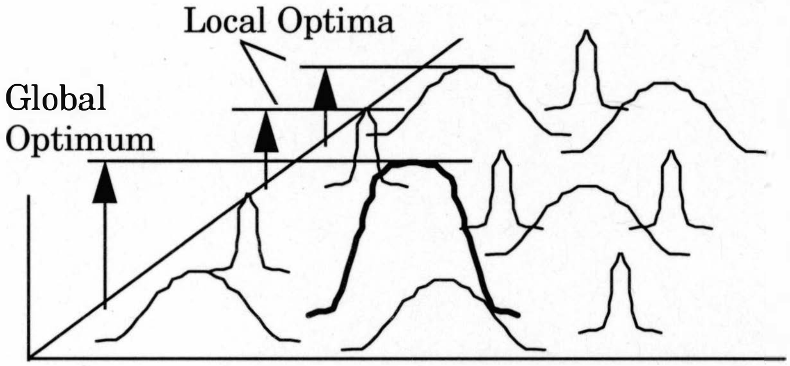


Figure 6. Landscape of Solution Space Illustrating Local and Global Optima.

The Evolutionary Strategy

The evolutionary strategies were developed as a more problem dependent method, beginning primarily with floating point numerical function optimization problems. This is where the chromosome took on forms more specific to the solution than the binary string. The operator most typically used in evolutionary strategies was the mutation operator and this operation traditionally preceded selection, thereby allowing parents and children to compete. In fact, a percentage of the current population is fully duplicated and all operations occur on this offspring. Finally, the 'best' chromosomes are selected for the next generation. (Refer to Figure 7.) This method of selection is deterministic in nature so the bad chromosomes have no chance of survival. The parent chromosomes are also retained such that a duplicated offspring does not replace it in the event its fitness degrades.

Variable Population

At first it seemed that population size was merely a variable to be

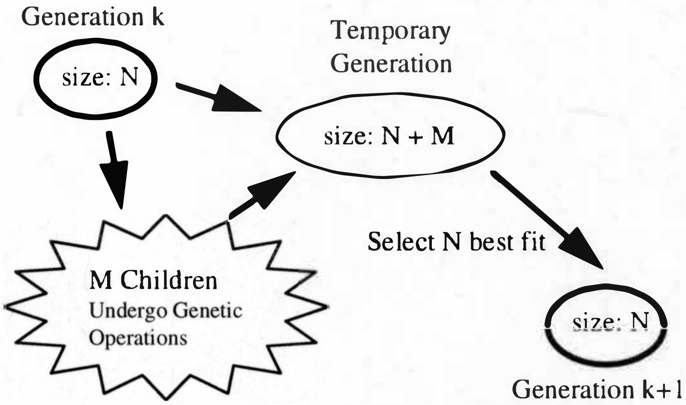


Figure 7. The Operation/Selection Cycle of the Evolutionary Strategy.

manipulated in either of the genetic algorithm or the evolutionary strategy approaches. However, in actuality, the variable population/life strategy becomes a unique method since the selection criteria is replaced by an age calculation. The underlying philosophy of this concept is that while the power of evolutionary algorithms lies in their ability to evaluate a population worth of chromosomes simultaneously, there is no need to indefinitely retain and evaluate relatively poor solutions.

Therefore, this method is inherently self adapting in that the population size varies as the run progresses. In any particular generation, new chromosomes (created randomly from existing chromosomes) are assigned a life expectancy (age). This age is initially calculated based on the chromosome's fitness relative to the current generation as well as all chromosomes evaluated thus far. In each subsequent generation, the age is decremented and finally the chromosome is removed when the age is zero. 'Survival of the fittest' is accomplished by assigning longer life expectancies to the better chromosomes thereby keeping them around longer to be the parents of (more of) the randomly created new chromosomes.

The genetic algorithm with Varying Population size (GAVaPS) as described in [1], compared three different age calculations, one of which is based merely on the fitness relative to the current generation (typical of the genetic algorithm). The second age computation is based on the chromosome's fitness relative to all solutions created thus far. This method has a drawback when many chromosomes have high fitness: the assigned ages are all high and the population grows unnecessarily large. Finally, the authors derived a hybrid calculation which looks at whether the fitness is above or below the average of the current generation, then linearly determines age within the upper or lower half of the current generation based on that result.

Simulated Annealing

"There is a deep and useful connection between statistical mechanics (the behavior of systems with many degrees of freedom in thermal equilibrium at a finite temperature) and multivariate or combinatorial optimization (finding the minimum of a given function depending on many parameters)." [19, page 1] Because of this connection, Kirkpatrick went on to demonstrate the application of the simulated annealing characteristics to the optimization of complex problems. Here, simulated annealing will be viewed in the light of evolutionary algorithm techniques.

This method holds and evaluates one allocation (schedule) at a time. Viewing this allocation as a chromosome, a mutation-type operator is applied to create a new chromosome. If the new chromosome is better than the original, the new replaces the original. However, if the original is better, the new replaces the original with a probability that exponentially decreases as the run progresses. The specific probability used in this work is discussed more fully in the next section.

List Scheduling

Greedy by nature, this algorithmic method creates an allocation of tasks deterministically. Simply, all tasks with preconditions satisfied are placed in an available pool. A timing mechanism monitors processors until one is ready for a new task. The available pool is searched for the "best" task which is then assigned to the open processor. The timing mechanism continues simulating the start-completion of tasks on the processors with the available pool consistently updated based on satisfying pre-conditions. The algorithm stops when all tasks have been assigned.

The tailoring of this algorithm to problem instances is mainly found in the search/selection technique. The original list scheduling algorithm, developed by R. L. Graham [10], used a criteria that chose the task having the longest execution time.

Since then, many other search/selection methods have been developed for significantly more complicated systems. Such systems may have resources (processors) that perform limited operations, complex precedence constraints between the tasks and even real-time scheduling [13], [17], [20].

TECHNIQUES USED IN THIS EFFORT

The Problem - Burg's Algorithm

To perform a comparison of different problem solution techniques, one problem must be chosen. In this effort, an algorithm known as Burg's Algorithm is used. The purpose of Burg's is to process signals in many applications such as speech processing and spectral analysis. The algorithm accomplishes this by fitting an autoregressive model to a time series data set.. The desired autoregressive model would appear as $x_n + (a_1)^m x_{n-1} + \ldots + (a_m)^m x_{x-m} = (e_n)^m$ where x_n is the autoregressive process, $(a_1)^m$ through $(a_m)^m$ are the process parameters and $(e_n)^m$ is white noise. The purpose of the algorithm is to estimate the coefficients $(a_i)^m$. The algorithm uses the fact that the model can be of two different forms where the autoregressive coefficients a_i's (of direct form II) are related to the reflection coefficients c_i's (from the lattice structure form). Based on this, the forward and backward prediction errors are estimated and finds the optimal choice of the reflection coefficients is:

$$c_{m+1} = -2\left(\sum_{n=m+2}^{M} e_n^n b_{n-m-1}^m \right) / \left(\sum_{n=m+2}^{M} (e_n^m)^2 + (b_{n-m-1}^m)^2 \right)$$

In [25], the authors further develop this into sequential algorithm pseudocode as shown in Figure 8 and proceed to lay out a maximally parallel graph for a third order model (MAX=3) of a data series containing five data points (M=5) as shown in Figure 9. The algorithm for this prob

```
INITIALIZATION
    for i = 1 to M do
        $e(i) = x(i)$
        $b(i) = x(i)$

THE MAIN LOOP
    for n = 1 to MAX do
        $s1 = 0.0$
        $s2 = 0.0$

        for i = n+1 to M do
            $s1 = s1 + e(i) * b(i-n)$
            $s2 = s2 + e(i)^2 + b(i-n)^2$

        $c(n) = -2.0 * s1/s2$

        if n > 1 then do
            for i = 1 to n-1 do
                $a1(i) = a(i) + c(n) * a(n-i)$
            for i = 1 to n-1 do
                $a(i) = a1(i)$

        $a(n) = c(n)$

        for i = 1 to M do
            $temp = e(i) + c(n) * b(i-n)$
            $b(i-n) = b(i-n) + c(n) * e(i)$
            $e(i) = temp$

WHERE
    a's are the autoregressive coefficients
    e's represent white noise
    c's are reflection coefficients
    b's are used to compute forward and backward prediction
        errors
```

Figure 8. Sequential Burg Algorithm for M Data Points and MAX Reflection Coefficients.

lem instance requires 27 different tasks which are numbered as if performed sequentially. As shown in the graph, all tasks presented on the

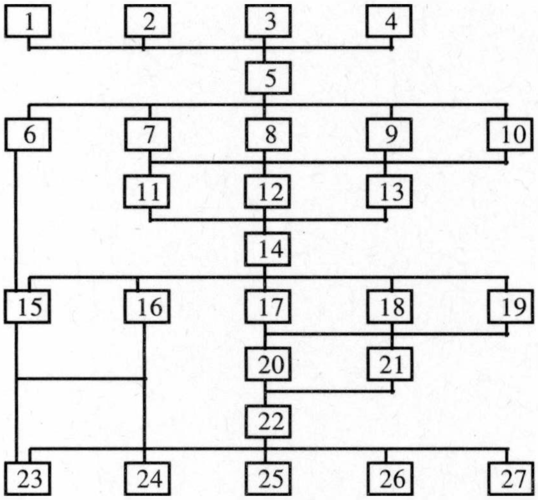


Figure 9. Burg Algorithm Depicting Task Graph for 3rd Order Model to Fit Five Data Points

same level may be performed in parallel and tasks on levels above must be performed before tasks on levels below.

To evaluate a more realistic example, the problem size was increased to a third order model of 1,024 data points resulting in 6,140 tasks. Referring again to Figure 8, the 1st level of the plot in the larger problem instance has 1,023 tasks which must be completed before task number 1,024. Therefore, the task scheduling problem here is to assign 1,023 tasks to two processors in such a way as to have the quickest execution time before proceeding to task 1,024.

Implementation Notes

Fitness and Cost

Since the purpose of this work is to compare methods of solving a problem, objective measurements must be established. To prove the benefits of any particular method over all others would include establishing the best result for the minimum cost. The most commonly measured result of parallel implementations versus sequential ones is speedup, therefore, that is the comparable outcome. Speedup is defined to be the actual schedule length of the sequential implementation divided by the schedule length of the parallel implementation. Realizing that in the worst situation, the parallel case would not improve the sequential at all and speedup would be one. In the best situation if there were two processors versus one, the parallel schedule length would be half the sequential and speedup would equate to two, the number of processors used. The most apparent cost, ignoring initial programming overhead, is the latent computation time (the elapsed time from program start to program end).

To ensure the integrity of the speedup/computation time results, one hundred runs were performed of each method. The conclusions section of this paper provides the summary details of the runs of the various methods.

General

All problem solution techniques were coded in C++ and compiled with the Gnu g++ compiler in the UNIX operating environment. To the greatest extent possible, code was duplicated throughout the various methods to provide comparable results. The discussion of individual techniques points out those instances where duplicating code was not possible.

At the onset of the project, the sequential algorithm was implemented to record exact time requirements for every task in the 6,140 task program. These experimentally determined times were then used as input to the program so that task allocation/execution would simulate actual times. Many tasks performing the same function (i.e. simple addition) that proved to require approximately the same time were averaged and assigned equal times.

The data structure used for the chromosome design is nearly identical in all methods described as are the mutation operators. Therefore, these are defined separately. The remaining details, such as selection criteria, generational transition, probability of the use of operators and stopping conditions vary with each method and are laid out in each method's individual section.

The Chromosome Data Structure

As stated previously, the data structure representing the chromosome should encode all relevant problem parameters. In this work, the chromosome, illustrated in Figure 10, consists of:

1. Two integer arrays encompassing the allocation: each array holds the specific task numbers to be executed on one of the two processors. The order of the tasks in the array is the order the tasks will be executed on the processors (left to right as pictured will be first to last when executed).

2. The schedule length: time required for the allocation represented in (1).

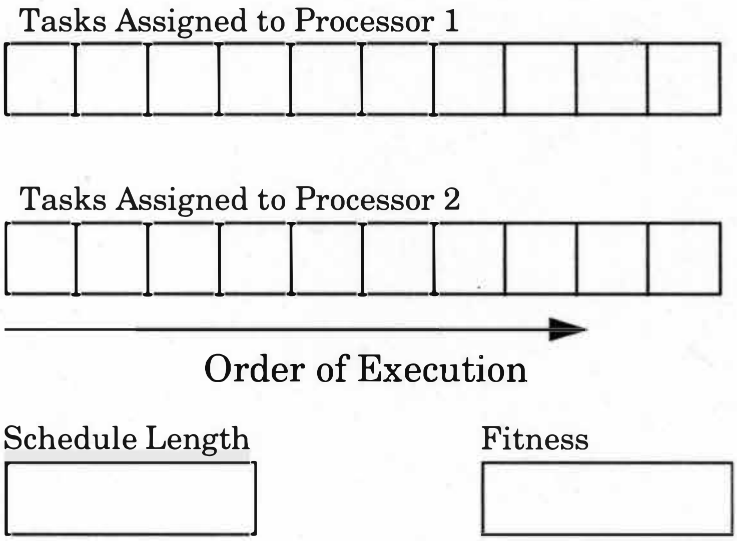


Figure 10. The Chromosome Data Structure.

3. The fitness of the chromosome: defined as the sequential schedule length divided by this (chromosome's) allocation schedule (2).

The two arrays that hold task assignments for the two processors do represent the entire solution, while maintaining the schedule and fitness allows for quick comparisons of chromosomes (recalling there are 6,140 tasks in the schedule). Additionally, the integer arrays provide for easy manipulation by operators.

Referring back to the example given in the Introduction section, Figure 11 shows repeats the task graph and illustrates the chromosome data structure just described.

The Operators

The two most common operators in evolutionary techniques are recombination and mutation. Looking at this problem, a schedule of distinct tasks on two processors, it is apparent that the recombination of two

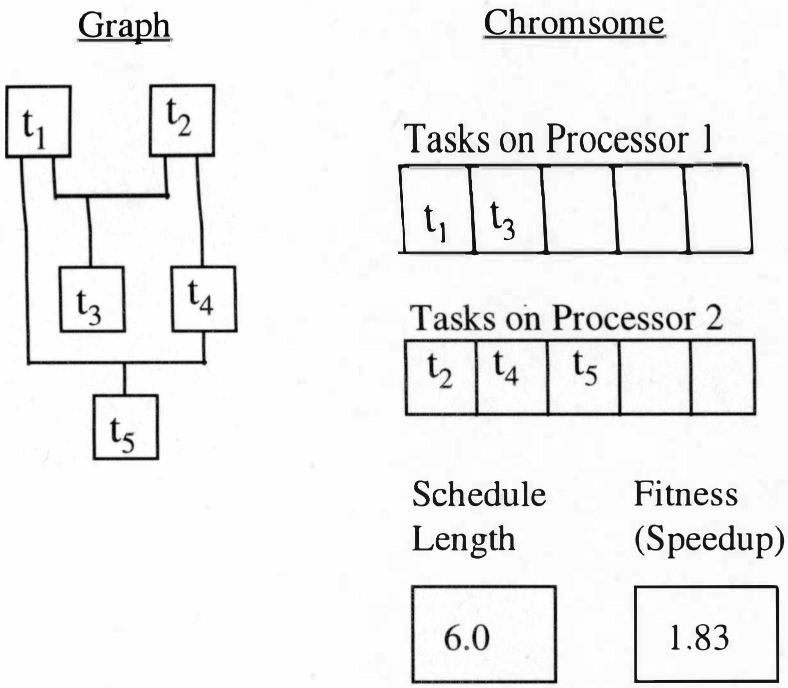


Figure 11. Example Showing Chromosome Data Structure.

chromosomes (task allocations) does not create two complete schedules. Consequently, the operators created for use in this problem are referred to as Intra-Processor Mutation and Inter-Processor Mutation (Intra-PM and Inter-PM).

Intra-Processor Mutation performs as the typical mutation operator does. In this case mutation consists of: (a) one of the two processors is randomly chosen, (b) two candidate task locations are chosen randomly, and (c) the tasks in the two locations (in the selected processor) are swapped. Figure 12 presents an example of the Intra-PM operator on the chromosome data structure used here.

To produce an effect similar to the typical recombination operator, the Inter-Processor Mutation operator was developed. This Inter-PM operator proceeds by: (a) noting the number of tasks on each processor,

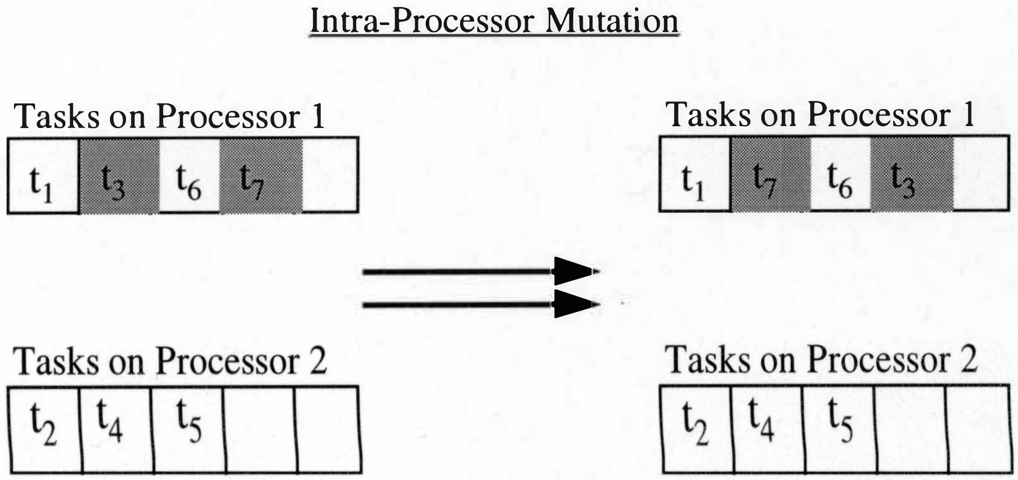


Figure 12. Example of Intra-PM Operation Within One Chromosome.

(b) locating a crossover point (randomly) on each processor, (c) determining the number of tasks to be moved and the direction of the move, and (d) reassigning tasks from one processor to the other. Unless specified otherwise, the number of tasks moved from one processor to the other is a randomly generated number between five and ten, inclusively.

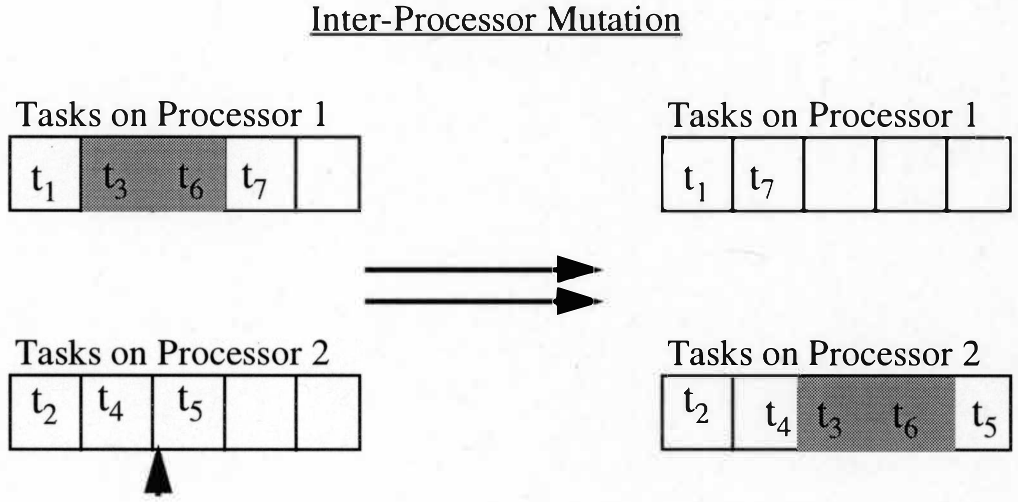


Figure 13. Example of Inter-PM Operation Within One Chromosome.

In the methods that use the Inter-PM and Intra-PM operators, the following fact applies. If the chromosome newly created by the operator results in an infeasible allocation (primarily due to processor deadlock caused when precedence constraints are violated), the chromosome is returned to the allocation it held before the mutation.

List Scheduling Details

To have a broad comparison of the evolutionary techniques to other proven solution derivations, one of the popular methods, list scheduling was chosen. List scheduling is a greedy algorithm where, in general, at every iteration, the very best item is selected. In the scheduling problem, ready tasks, defined as those tasks whose precedence constraints are accomplished, are placed in a waiting pool. The execution of tasks on the two processors is simulated such that when a task completes, the available processor looks to the pool of ready tasks to select the most appropriate task. In the simple case (presented by Graham in [10]), the task requiring the longest amount of time to complete is chosen, thereby removing it from the ready pool and assigning it to the available processor. After every such assignment, the simulated processor clocks are updated and the precedence constraints of all tasks are again evaluated to determine if additional tasks can be placed in the ready pool. This execute-select cycle continues until there are no remaining tasks to be performed.

Unique implementation details for this method are minimal. The two integer arrays holding task allocations are used, and obviously the schedule and fitness are maintained. List scheduling creates one allocation, task by task, therefore, there is no use of operators nor generational transitions from one population of solutions to another.

Simulated Annealing Details

Moving up one step into heuristic techniques leads to the simulated annealing method. In this case, the chromosome data structure is used, and the Inter-PM and Intra-PM operators are involved. Refer to [19] and [21] for detailed descriptions of the mechanical basis of simulated annealing and the application to optimization problems. The annealing technique is adapted from the statistical mechanics field. A typical application is to find the low temperature state of a material. An example is growing a single crystal. At first, melt the substance, then lower the temperature slowly, spending a long time at temperatures in the vicinity of the freezing point. The temperature changes are scheduled to allow sufficient time spent stabilizing at each temperature to prevent the material from forming only a glass and not a crystal (the optimum).

When this method is used in optimization problems, the temperature change schedule is implemented as a probability that decreases exponentially with time. Solutions stabilize, however, relative to the probability, fluctuations are introduced. As time passes, fewer fluctuations are allowed and an optimal solution has been formed.

The simulated annealing optimization algorithm used here is summarized in Figure 14. In an iteration, a new chromosome is created by first duplicating the current chromosome, then performing either the

Intra-PM or Inter-PM operator. Once created, the new chromosome's fitness is compared to the current chromosome's fitness and if better, the new will always replace the current. However, when the new chromosome's fitness is less than the current, it also may replace, but only at a probability less than given by: e^{-Tn}. The n represents the time, or iteration in the process and therefore provides a significant decreased probability as time is extended greatly. The T is a carry over from the temperature in the mechanical model, here it was set at 0.000690. Notice, only one chromosome is maintained from generation to generation.

The number of generations was set to 10,000, to perform a search lasting approximately one hour. The constant, T, was calculated based on the probability (e^{-Tn}) of a worse chromosome replacing a better to be only

```
for n = 1 to MAX_GEN do

    temp_chrom = current_chrom

    if random_number <= Probability_Mutate then
        perform Intra-Processor Mutate on temp_chrom
    else
        perform Inter-Processor Mutate on temp_chrom

    if fitness(temp_chrom) > fitness(current_chrom) then
        current_chrom = temp_chrom
    else
        No_Bads_Accepted = $exp^{-Tn}$
        if random_number <= No_Bads_Accepted then
            current_chrom = temp_chrom

end (* for MAX_GEN *)
```

Figure 14. Simulated Annealing Algorithm Using Chromosome and Operators From Evolutionary Techniques.

0.001 at the end of the search (n=10,000). This probability proved the best when compared to 0.01 and 0.001 for the same search time. The probability of Intra-Processor Mutation was set to 20%, thereby allowing 80% of newly created chromosomes to have the Inter-Processor Mutation operator applied (using the Inter-PM + Intra-PM = 100% general rule).

The phenomena of allowing a newly created chromosome, that is not an improvement (based on fitness value), to replace the current chromosome is intended to move the search around the landscape of solutions, hopefully landing on the hill with the global optimum (refer back to Figure 6).

Genetic Algorithm Details

The genetic algorithm is array based and implemented according to Figure 15.

To begin, a roulette wheel is created by finding the sum of all fitness in the population, then for each chromosome, calculating a percentage fitness as the ratio of its respective fitness to that total. Fill the slots with chromosomes as shown in the example in Figure 16. When selecting a chromosome from the roulette wheel, generate a random integer number between 0 and 100%. Find that slot in the roulette wheel (or number line) and retrieve the chromosome for duplication.

Once created, a new chromosome will undergo Intra-PM operation with a probability of 10% or Inter-PM with a probability of 80%. This Inter-PM value and the population size of 25 were determined experimentally [11]. Tests compared the effects of Inter-PM settings of 40%, 60%

and 80% and in all cases, the 80% value produced higher speedup values. In the genetic algorithm, the general rule of Inter-PM + Intra-PM = 100% is overruled to allow some number of chromosomes to go into subsequent

```
CONSTRUCT ROULETTE WHEEL BY:

    1. find total fitness of each chromosome in population
    2. compute the ratio of the fitness of each chromosome
          to the total fitness
    3. assign slots in number line based on percentage fitness

CREATE NEW POPULATION:

new_chrom[1] = best of(current_chrom)

for index = 2 to Population_Size do

    (* find chromosome to duplicate *)
        pb = random_number
        find location, j, of pb in number line
        temp_chrom = current_chrom[j]

    case random_number_2
        < Prob Inter-PM:
            perform Inter-PM on temp_chrom
        > Prob Inter-PM < (Prob Inter-PM + Prob Intra-PM):
            perform Intra-PM on temp_chrom

    new_chrom[index] = temp_chrom

end (* for population size *)

UPDATE CURRENT CHROMOSOME ARRAY:

for i = 1 to Population_Size do

    current_chrom[i] = new_chrom[i]

end (* for population size *)
```

Figure 15. The Genetic Algorithm.

Percentage Fitness

Chromosome 1	12
Chromosome 2	26
Chromosome 3	8
Chromosome 4	32
Chromosome 5	22

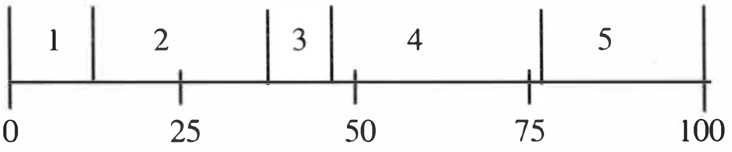


Figure 16. Example of Loaded Roulette Wheel.

generations unchanged. Therefore, Intra-PM was held to 10% with the 80% Inter-PM probability. Additionally, a comparison of two population sizes, 50 and 150 proved no appreciable difference in speedup results, therefore, the small number of 25 was chosen to minimize memory overhead. To prevent the best chromosome from being lost, the additional step to load it into the new chromosome array is included.

The second case of the genetic algorithm includes a ‘self adapting factor.’ This factor is applied to any chromosomes (other than the ‘best’ chromosome which is always included in the new generation) that have a fitness value equivalent to the best schedule within that particular generation. It is used in conjunction with the Inter-PM operator by limiting the number of tasks moved from one processor to the other. This is utilized since the best schedules will generally have a reasonably ‘balanced’ allocation of tasks (especially in the absence of precedence constraints). If too many tasks are moved from one processor to the other, the balance is significantly thrown off. The number of tasks moved is therefore held between two and five while the rest of the chromosomes undergoing Inter-PM remain between five and ten tasks shifting.

The stopping condition for both cases of the genetic algorithm is to monitor the best fitness value for every generation. If the fitness does not improve after 20 generations, stop the process and report the results.

Evolutionary Strategy Details

The evolutionary strategy is also array based. The transition from one generation of chromosomes to the next is depicted in Figure 17. Specifically, the chromosome data structures is the same as in the other methods. The Inter-PM and Intra PM operators are applied to every chromosome in the temporary array of duplicated chromosomes at probability rates of 80% and 20% respectively. These rates were determined experimentally by varying the two rates, however the two always combined to 100%. The population size of 25 was also used with the evolutionary strategy.

As shown in Figure 17, the best chromosome(s) will always survive since the selection criteria is deterministic. In the case of the self adapting factor, the best fitness is found and all chromosomes having that fitness have the limited number of task switch in the Inter-PM operation, namely between two and five. All other chromosomes switch between five and ten tasks during Inter-PM.

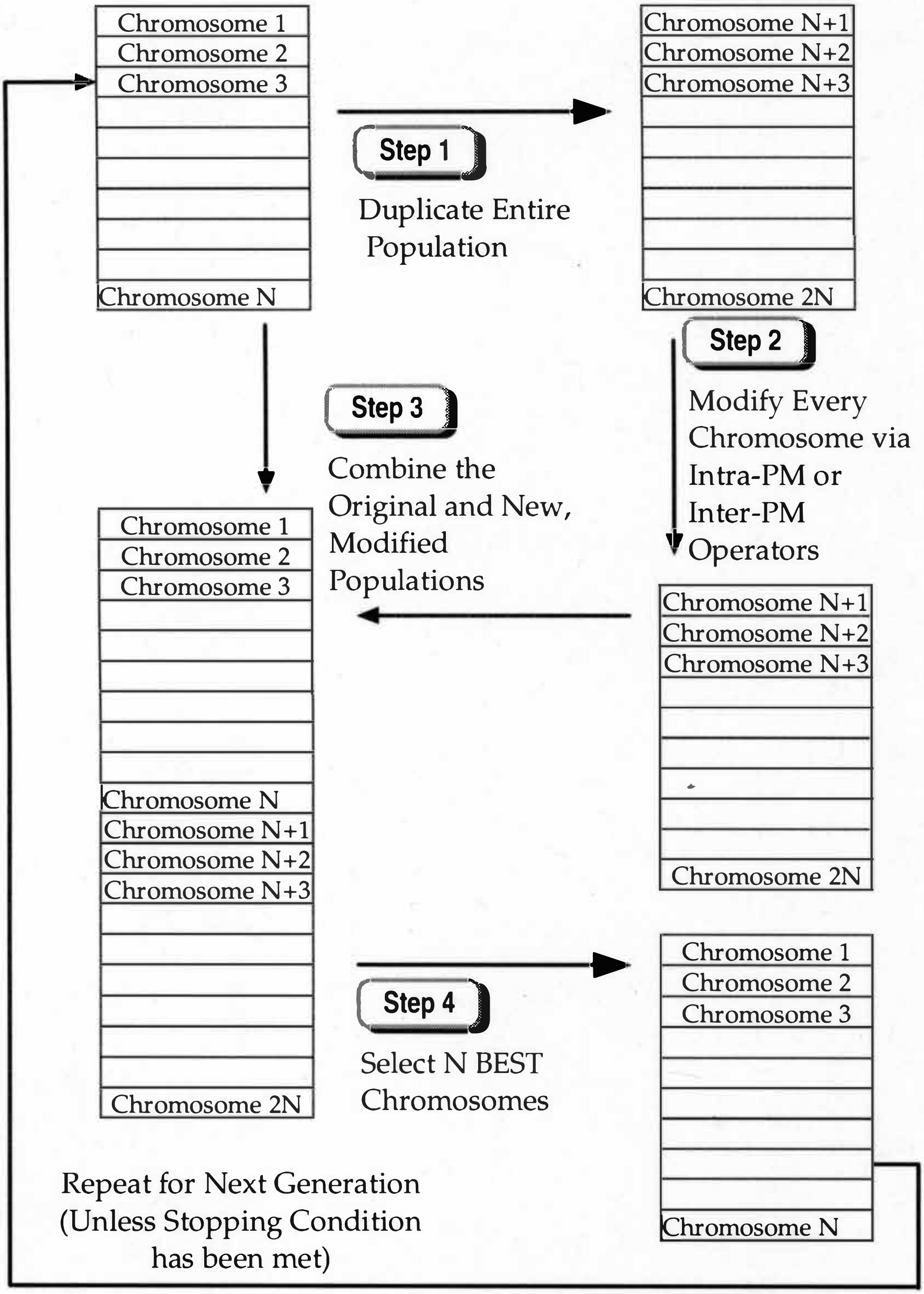


Figure 17. Evolutionary Strategy Generational Transition.

The stopping criteria here is set the same as the genetic algorithm: if, after 20 generations, there is no improvement in the best fitness, halt.

Variable Population Details

The variable population method was developed to minimize the number of chromosomes present in generations and thereby reduce the number of evaluations made throughout the search. To accomplish this, the chromosomes are given an age upon creation. This age is based on the chromosome's fitness relative to the others in the current population, as well as all the chromosomes created since generation one, and is specifically between a minimum and maximum age value. A new generation is started by reproducing ρ% of the current generation chromosomes and performing Inter-PM and Intra-PM operations on the new chromosomes and, again, an age value is assigned to each one. The last step in the generational transition to reduce the age of all chromosomes in the current population and remove any whose age has reduced to zero. This process allows the population to grow as 'good' chromosomes are created (and have high age values assigned) and decrease when 'bad' chromosomes are created and need only be kept around a short time (so low age values are assigned). The calculation of the age is a strategic part of the process. The authors of the GAVaPS [1] developed and tested three life strategies: (1) proportional, which assigns the age value proportional to the chromosome's fitness; (2) linear, which linearly interpolates the age span based on chromosome's fitness in current generation; and (3) bi-linear which attempted to sharpen the difference between lifetime values of nearly-the-best, yet still taking into account the maximum and minimum fitness values found thus far.

The three strategies taken from [1] were tried; however their performance tended to cause the population to grow very quickly (> 1,000 chromosomes in approximately 20 generations). In this problem, the chromosomes often have the same fitness value, many chromosomes would likely get the highest age value unnecessarily. Therefore, a fourth strategy was developed. The intent was to give those chromosomes having fitness greater than average (with respect to the current generation), an age from within the top third of the range of age values. Those chromosomes having fitness less than average, would be assigned age values from the lower two-thirds of the age range. The life calculation used is given in Figure 18.

The variable population implementation in this work is a linked

```
if fitness <= AvgFit then
    age = MinLT +((MaxLT-MinLT)/3 *
        ((fitness-MinFit)/(AvgFit-MinFit))
else
    age = MinLT + ((MaxLT-MinLT)/3) * 2 *
        ((fitness-AvgFit)/(AbsFitMax-AvgFit))

WHERE
    fitness = fitness of chromosome
    AvgFit = average fitness of chromosomes in this
      generation
    MinLT = minimum life value assigned
    MaxLT = maximum life value assigned
    MinFit = minimum fitness value in this generation
    AbsFitMax = maximum fitness for all generations
```

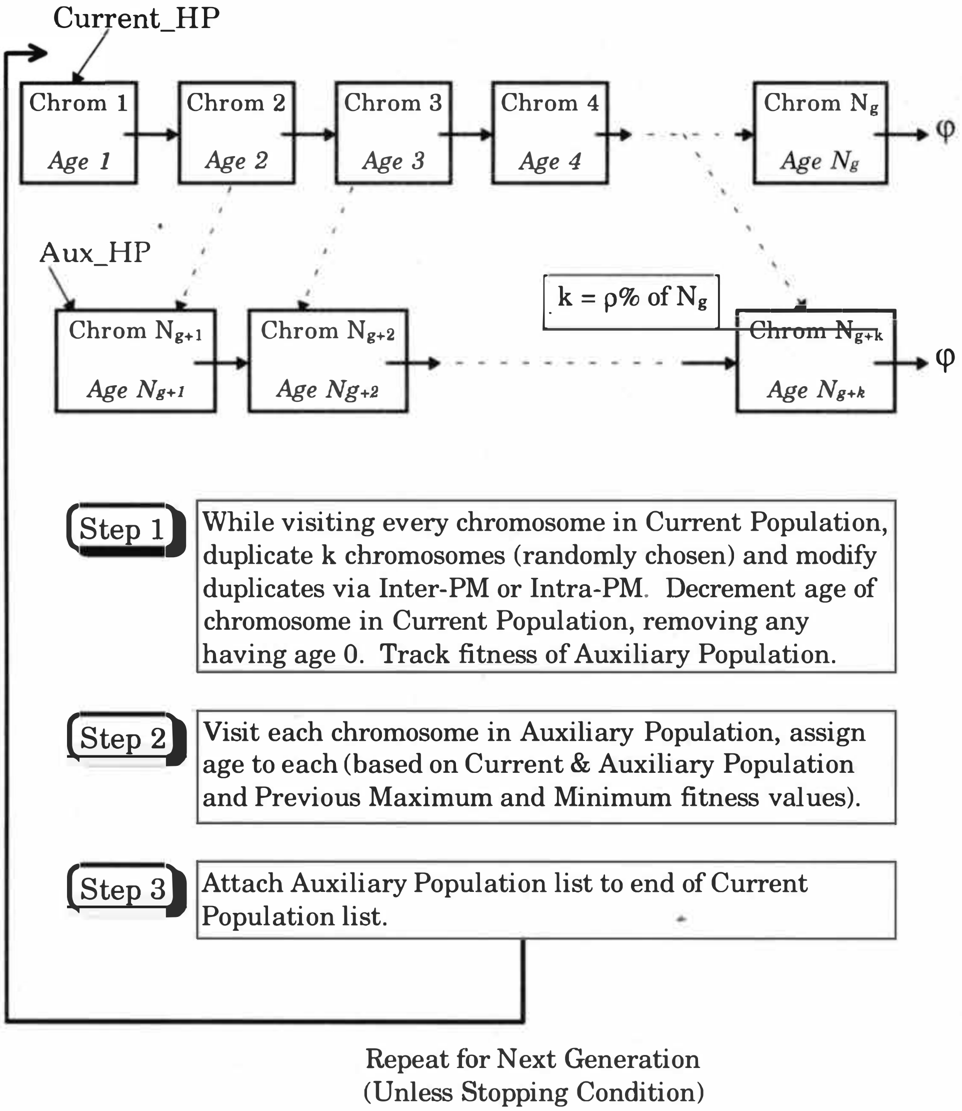
Figure 18. Life Strategy Calculation in Variable Population.

list, as shown in Figure 19, to accommodate the unknown maximum population size. The pointer to the current generation, Current_HP, indicates a population of N_g. The additional chromosomes created, pointed to by Aux_HP, total, k which is ρ% of the current population size.

Specific parameters used included a minimum life of one generation and maximum life of seven. Every process began with 20 chromosomes and the probability of duplication into the auxiliary population was ρ = 40%. These values were taken from the original GAVaPS work [1] and some variations of each were tested but found less desirable.

All chromosomes created (in the auxiliary population) undergo the operators: Inter-PM at the rate of 80% or Intra-PM at a rate of 20%. If a new allocation results in an infeasible schedule (processor deadlock due to violation of precedence constraints) the chromosome age is set to zero and the chromosome is removed from the auxiliary list before becoming part of the current list. The stopping criteria is the same as the other evolutionary methods, twenty generations without improvement in best fitness value stops the system.

An improvement was implemented that saved the best chromosome similar to the genetic algorithm approach. When current generation chromosomes undergo the age decrement, if the chromosome happens to be the best, it will not have its age reduced. Eventually, another chromosome will likely become the best. This allowed normal chromosome life-death spans and prevented early extinction of the population so the 20 generation convergence was always achieved.



Step 1: While visiting every chromosome in Current Population, duplicate k chromosomes (randomly chosen) and modify duplicates via Inter-PM or Intra-PM. Decrement age of chromosome in Current Population, removing any having age 0. Track fitness of Auxiliary Population.

Step 2: Visit each chromosome in Auxiliary Population, assign age to each (based on Current & Auxiliary Population and Previous Maximum and Minimum fitness values).

Step 3: Attach Auxiliary Population list to end of Current Population list.

Figure 19. Generational Flow of Variable Population Method.

CONCLUSIONS

Recall the major purpose of task scheduling is to determine a fast schedule quickly. Each method was tested by performing one hundred runs on a 70 Mhz Sparcstation 5 with 32 MB memory and the Solaris 2.4 operating system. The results are plotted as the Speedup benefit versus Time cost. Obviously, it is desirable to have data points in the upper left quadrant of such a plot. Looking at Figure 20 (median values of 100 runs except VPB - Variable Population, Best, which had 56 runs), allocations represented by data points in the lower left quadrant are not fast schedules, but are arrived at quickly. Those in the lower right quadrant have poor speedup results and take a long time to derive. Finally, those in the upper right quadrant have meritable speedups, however they may require a long time to find. This could be acceptable in the static scheduling realm where an allocation is derived one time then used often. A small gain in speedup that is realized often may be well worth the overhead spent once to find the allocation.

The reader should note that the quadrants just described are relative to the scale of the plot. Figure 20, the comparison of all methods, is scaled based on their relative median values. However, the remaining plots depicting individual methods are all on an identical scale, yet different from that of Figure 20. Bearing that in mind, keep the quadrants in perspective as the discussion continues.

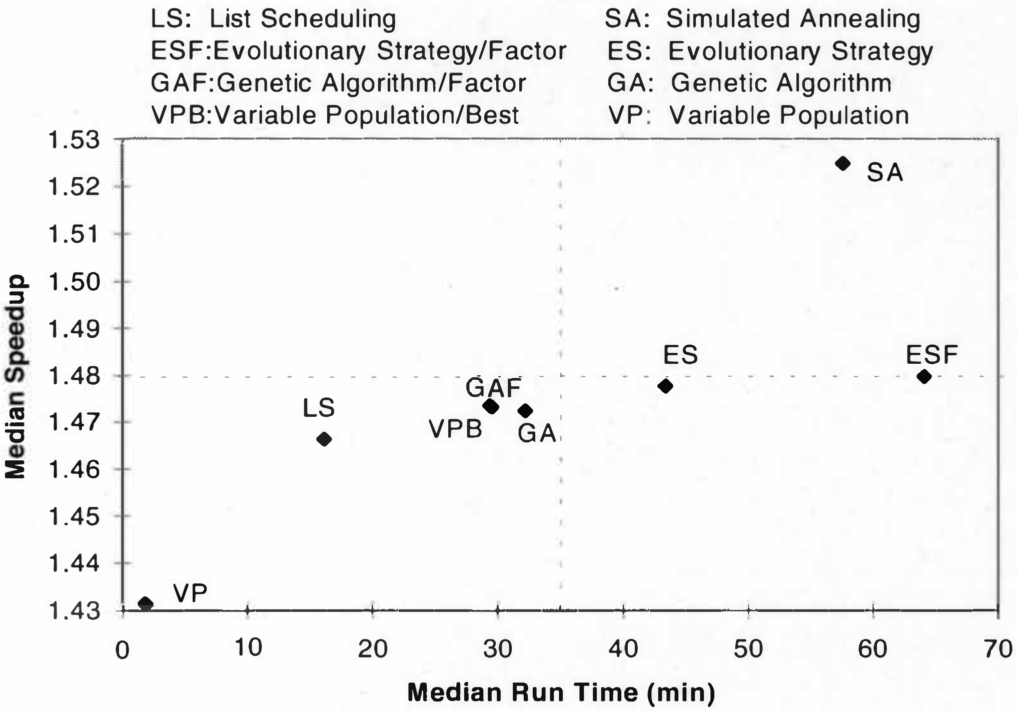


Figure 20. Results of All Methods Compared.

Looking now at the details of the comparison, the list scheduling method performed very predictably (less than satisfactory). The speedup obtained was mediocre and found exactly the same value in approximately the same amount of time, in every run. Figure 21 illustrates the results.

The genetic algorithm (Figure 22) typically converged relatively quickly. This is likely due to the selection criteria where the better chromosomes have a higher probability of surviving to future generations. As the median values suggest, results can be obtained in as little as half an hour, although the better speedups require longer searches.

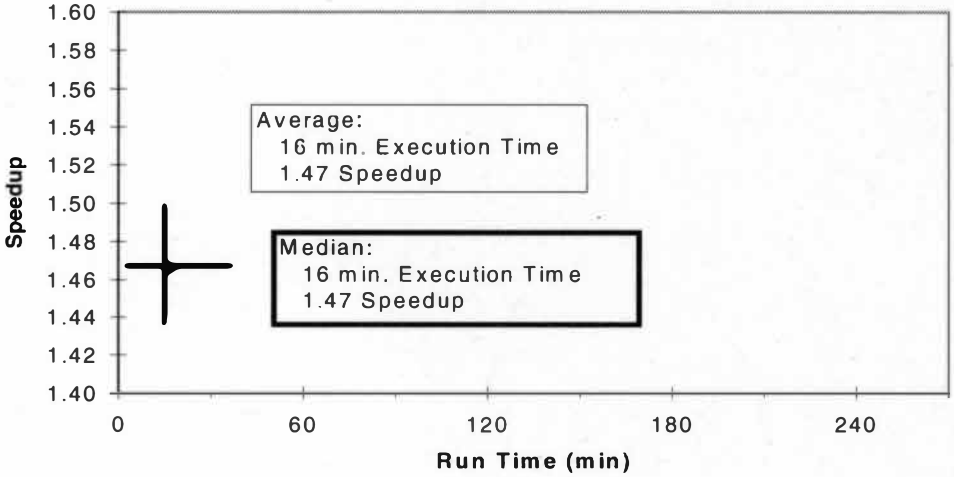


Figure 21. Results of List Scheduling.

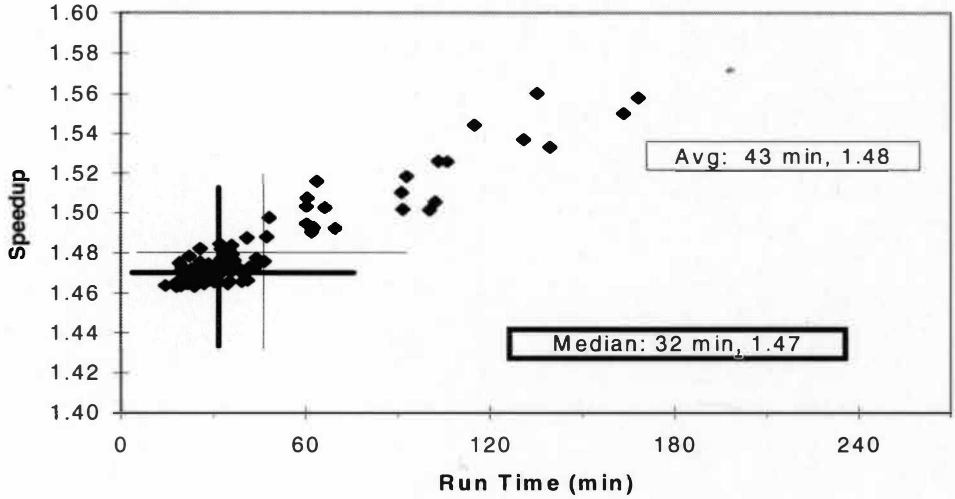


Figure 22. Results of Genetic Algorithm.

As expected, the evolutionary strategy's performance (Figure 23) closely resembled the genetic algorithm. The two methods are very similar

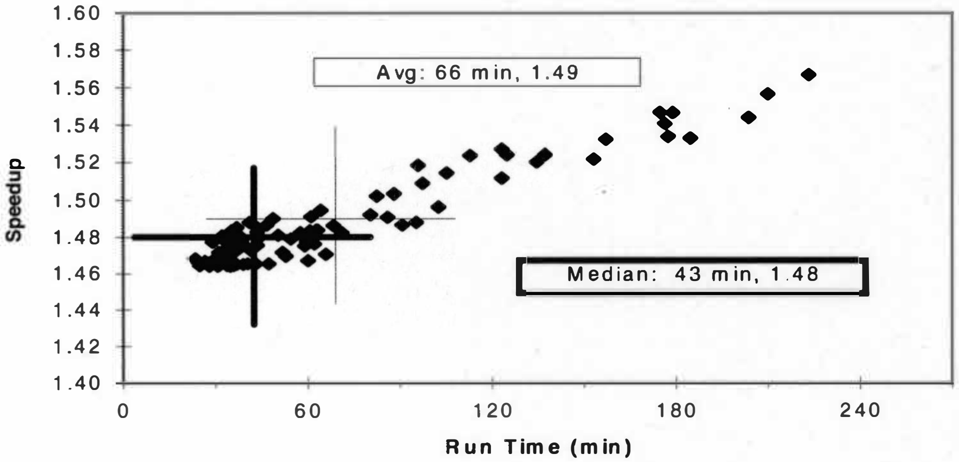


Figure 23. Results of Evolutionary Strategy.

in nature with the one difference primarily in the selection method. The deterministic approach enabled the evolutionary strategy to search longer before converging on a particular speedup value. Here the median indicates three-quarters to a full hour for the typical value which is slightly better than that produced in median time for the genetic algorithm. Modifying the number of tasks swapped in the Inter-PM operation ("the Factor") seemed to have a greater effect on the evolutionary strategy (Figure 24) than on the genetic algorithm (Figure 25) although the benefit of it is not obvious.

One significant trend found in both the genetic algorithm and the evolutionary strategy that is not present in the variable population method is that, typically, the longer the search, the better the allocation found. If there is time available for the overhead of finding the best schedule, one could simply increase the number of generations used to de

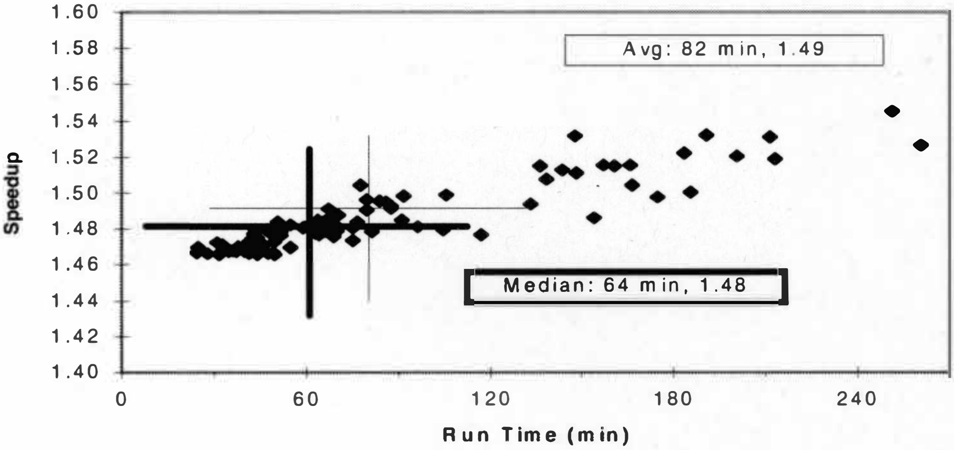


Figure 24. Results of Evolutionary Strategy With Self Adapting Factor (Inter-PM Operation on Best Chromosome in Population Only Moves 2-5 Tasks Instead of 5-10 Tasks).

tect convergence. In the variable population method, (Figure 26) however, more search time did not necessarily produce better schedules. Looking at the case where the best chromosome was not saved (Figure 27) the search died very quickly with very poor results. In fact, the process died early (never reaching 20 generations to converge on one result) 80% of the time and 500 runs were made to get 100 feasible runs. By saving the best, however, there were still quickly found poor results (convergence in minutes) and not too many good results after hours of searching.

Finally, the simulated annealing method (Figure 28) worked exceptionally on this problem instance. The maximum speedup obtained of all methods, 1.59, was with the simulated annealing method and the minimum speedups obtained with this method were typically at or above

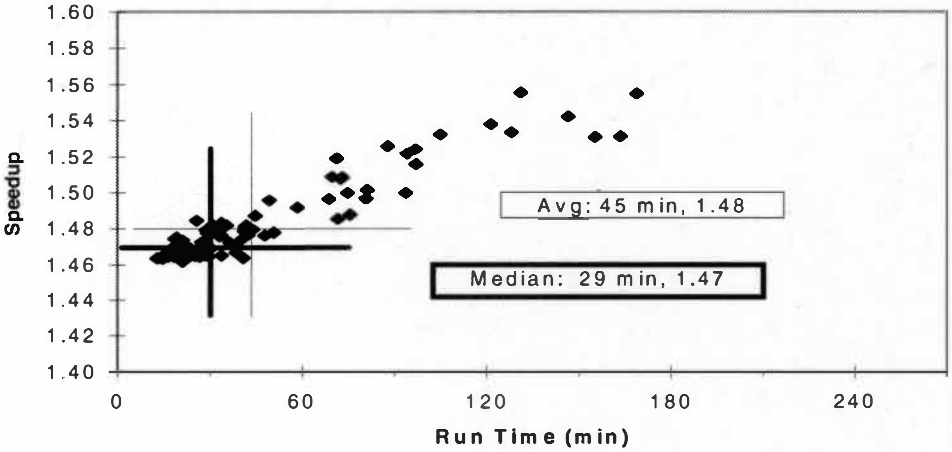


Figure 25. Results of Genetic Algorithm With Self Adapting Factor (Inter-PM Operation on Best Chromosome in Population Only Moves 2-5 Tasks Instead of 5-10 Tasks).

1.48. Simulated annealing consistently found high speedup values in almost exactly one hour labeling it 'a good bet.'

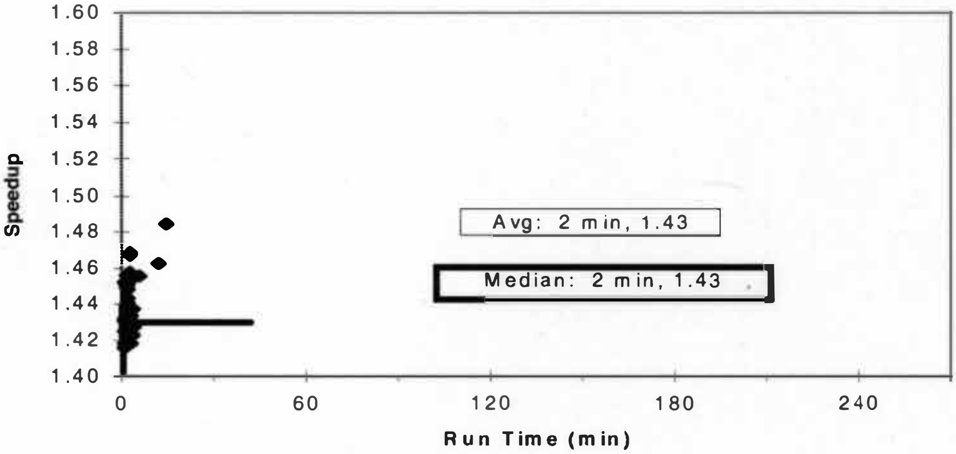


Figure 26. Results of Variable Population.

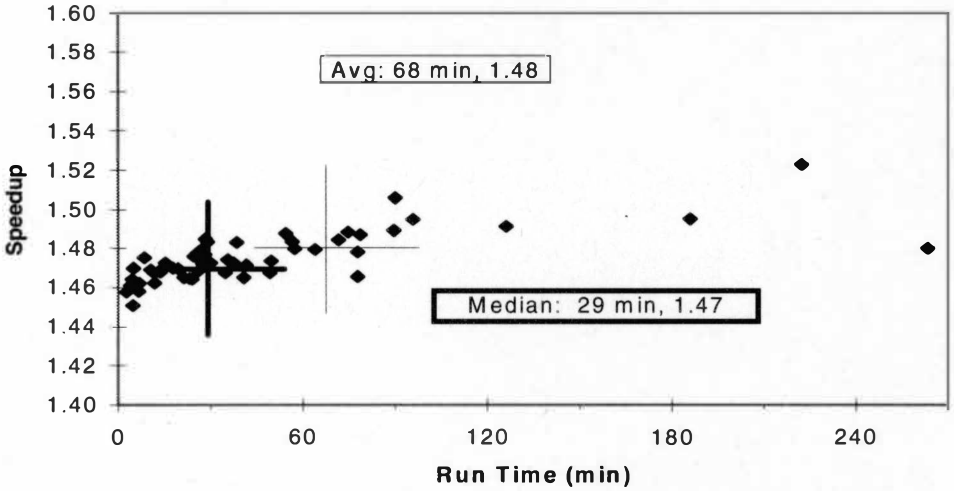


Figure 27. Results of Variable Population With Best Chromosome Saved.

The downfall of the genetic algorithm and the evolutionary algorithm as compared to the simulated annealing method appears to be the overhead. The actual number of evaluations was fixed for the simulated annealing method at 10,000 (one chromosome for 10,000 generations). The genetic algorithm and evolutionary strategies typically used 0.55-0.58 minutes to evaluate a generation of chromosomes. Therefore, all speedup results requiring more than one hour evaluated only about 2600 chromosomes. Those runs using upwards of three hours made over 8,000 evaluations and only those above four hours actually reviewed more than 10,000 allocations. This clearly states that if the genetic algorithm and evolutionary strategy were allowed to run through the same number of evaluations as the simulated annealing method, they too would produce high speedup results.

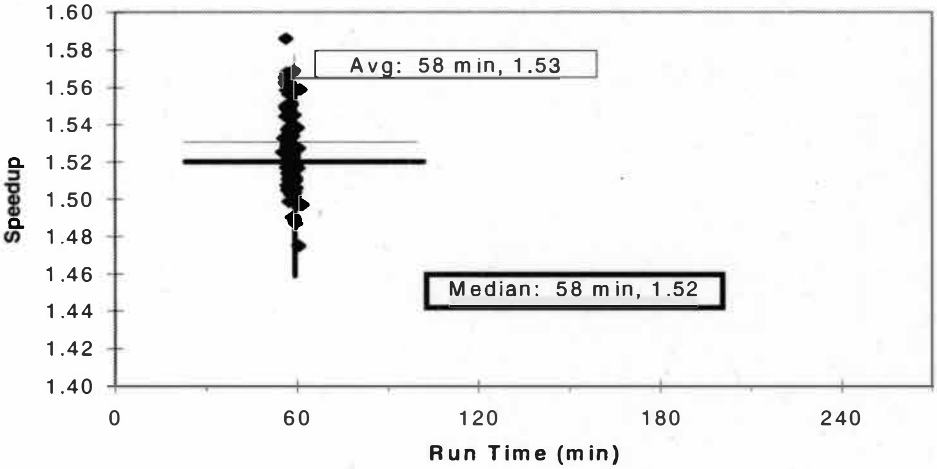


Figure 28. Results of Simulated Annealing.

It seems that in this problem instance, the chromosome data structure and mutation operators work very well with all three methods. It may be that a decision must be made on whether to review one chromosome at a time or an entire population. If the population overhead can be reduced, the potential for good results quickly is apparent. Ironically, the purpose of the variable population technique is exactly to reduce the overhead of carrying unnecessary chromosomes from one generation to the next. However, more development is obviously required to realize those benefits.

BIBLIOGRAPHY

[1] Arabas, J., Z. Michalewicz & J. Mulawka, “GAVaPS - A Genetic Algorithm with Varying Population Size,” Proceedings of the IEEE Conference on Evolutionary Computation, Volume 1, 1994, pp. 73-77.

[2] Bae, I. H. “A Systematic Genetic Algorithm for Task Allocation in Multiprocessor Systems”, Infoscience, 1993, pp. 663-669.

[3] Borriello, G. & D. M. Miles, "Task Scheduling for Real-Time Multi-Processor Simulations," Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software, 1994, pp. 70-73.

[4] Casavant, T. L., & J. G. Kuhl, “A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems,” IEEE Transactions on Software Engineering, Vol. 14, No. 2, February 1988, pp. 141-154.

[5] Filho, J. L. R., P. C. Treleaven & C. Alippi, “Genetic-Algorithm Programming Environments,” (IEEE) Computer, June 1994, pp.28-43.

[6] Galletly, J. E., “An Overview of Genetic Algorithms,” Kybernetes, Volume 21, No. 6, pp. 26-30.

[7] Garey, M. R. & D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.

[8] Gillies, D. W. & J. W. S. Liu, “Greed in Resource Scheduling,” Acta Informatica 28, 1991, pp. 755-775.

[9] Goldberg, D. E., Genetic Algorithms on Search Optimization, and Machine Learning, Addison-Wesley Publishing Co., 1987.

[10] Graham, R. L., “Bounds on Multiprocessing timing Anomalies,” SIAM Journal of Applied Mathematics., Vol. 17, No. 2, March 1969, pp. 416-429.

[11] Greenwood, Garrison, A. Gupta & K. Cousineau, "A Comparison of Evolutionary Techniques for Task Scheduling in Distributed Systems," (not published) May 1994, pp. 1-21.

[12] Greenwood, Garrison, A. Gupta & K. McSweeney, "Scheduling Tasks in Multiprocessor Systesm Using Evolutionary Strategies, Proceedings of the IEEE Conference on Evolutionary Computation, Volume 1, 1994, pp. 345-349.

[13] Gutberlet, P., H. Kramer, & W. Rosensteil, "CASCH - A Scheduling Algorithm for "High Level" - Synthesis," EDAC Proceedings of the European Conference on Design Automation, 1991, pp. 311-315.

[14] Han, B., "A Systematic Genetic Algorithm for Task Allocation in Multiprocessor Systems," Infoscience, 1993, pp. 663-669.

[15] Holland, J. H., Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, Michigan., 1975.

[16] Hou, E. S., R. Hong & N. Ansari, "Efficient Multiprocessor Scheduling Based on Genetic Algorithms," 16th Annual Conference of IEEE Industrial Electronics Society, Volume II, 1990, pp. 1239-1243.

[17] Hwang, J.-J., Y.-C. Chow, F. D. Anger, & C.-Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communications Times," SIAM Journal of Computing Vol. 18, No.2, April 1989, pp. 244-257.

[18] Hwang, K., Computer Architecture and Parallel Processing, McGraw Hill, Inc. 1984.

[19] Kirkpatrick, S. , C. D. Gelatt, Jr. & M. P. Vecchi, "Optimization by Simulated Annealing," Science, 13 May 1983, Volume 220, Number 4598, pp. 671-680.

[20] Liao, G., E. R. Altman, V. K. Agarwal, G. R. Gao, "A Comparative Study of Multiprocessor List Scheduling Heuristics," Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, 1994, pp. 68-77.

[21] Lin, S. C., & J. H. C. Hsueh, "A New Methodology of Simulated Annealing for the Optimsation Problems," Physica A 205, 1994, pp. 367-374.

[22] Lucasius, C. B. & G. Kateman, “Understanding and Using Genetic Algorithms, Part 1. Concepts, Properties and Context,” Chemometrics and Intelligent Laboratory Systems, 19, (1993), pp. 1-33.

[23] Michalewicz, Z., Genetic Algorithms + Data Strucures = Evolution Programs, Springer-Verlag Berlin Heidelberg, 1992.

[24] Nabhan, T. M., "Parallel Computations for a Class of Time Critical Processes, Ph. D. Dissertation, University of Western Australia,1994.

[25] Sammur, Nidal M., & M. Hagan, "Mapping Signal Processing Algorithms on Parallel Architectures," Journal of Parallel and Distributed Computing 8, 1990, pp 180-185.

[26] Srinivas, M., & L. M. Patnaik, “Genetic Algorithms: A Survey,” (IEEE) Computer, June 1994, pp. 17-26.

[27] Whitley, D., T. Starkweather & D. Fuquay, “Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator,” ICGA, 1989, pp. 133-140.