An incremental fitness function for partitioning parallel tasks

Annie S. Wu, Shiyuan Jin, Guy Schiavone
School of EECS
University of Central Florida
Orlando, FL 32816-2362
aswu@cs.ucf.edu, sjin@ist.ucf.edu, guy@pegasus.cc.ucf.edu

Kuo-Chi Lin
Institute for Simulation and Training
University of Central Florida
Orlando, FL 32826-0544
klin@pegasus.cc.ucf.edu

Abstract

We describe a novel GA approach to partition programs to be executed on a parallel system. Two unique features distinguish this GA from traditional GA programs. First, this GA uses a dynamically incremental fitness function which starts out rewarding for simpler goals, gradually increasing the difficulty of the desired fitness values or goals until a full solution is found. Second, this GA uses a flexible representation style which allows the GA itself more control over both the structure and the value of the evolved solutions.

1 Introduction

We successfully apply a genetic algorithm (GA) to the problem of partitioning a collection of tasks to be executed on a group of parallel processors.

Parallel programming improves the performance of a program by breaking the program down into smaller tasks and executing these subtasks simultaneously where possible. Executing the subtasks of a program in parallel can significantly speed up the run time of a program. The quality of a parallel algorithm can be evaluated with the following equation:

$$Speedup = \frac{Run\_time\_of\_best\_sequential\_algorithm}{Run\_time\_of\_parallel\_algorithm}.$$

Obviously, the larger the speedup, the better the parallel algorithm. Ideally, one hopes to achieve a maximum speedup of P when solving a problem using P parallel processors. In practice, such a speedup cannot be achieved for every problem for the following reasons:

- It is not always possible to decompose a problem into P tasks, each requiring $1/P$ of the total sequential time to execute.
- The structure of the parallel computer may impose restrictions that render the optimum running time unattainable (e.g. synchronization and communication overhead).

Many factors can affect the exploitation of parallelism in programs, including the partitioning of programs, the balancing of computational load among processors, and the overhead created by data communication between processors. The problem of partitioning a parallel program involves the specification of sequential units of a program that can be executed concurrently by the available processors such that the total parallel computation time of the program is minimized when inter-processor communication costs are included. The efficient partitioning of a program into parallel subtasks is not a trivial problem. (Bokhari 1981) has shown that there does not exist a polynomial algorithm to solve this problem.

Given the number of possible orderings of tasks in processors, the percentage of valid orderings can be very small. If a GA is not restricted to only work with valid individuals, the chance of randomly finding a valid ordering, let alone a good valid ordering, may be very low. Restricting a GA to only form valid individuals, however, may introduce unexpected biases in the system. In addition, such a strategy may require extensive revision of the system with each new problem.

We describe here a novel GA approach which can successfully partition programs to be executed on a parallel system. Our GA places no restrictions on the individuals that can be formed and does not require special operators or repair mechanisms to ensure validity. Rather, it attempts to give partial fitness for invalid individuals that contain some valid subsequences of tasks and encourages the formation of successively longer valid subsequences. Two unique features distinguish this GA from traditional GA programs. First, this GA uses a flexible representation style which allows the GA itself more control over both the structure and the value of the evolved solutions. Second, this GA uses a dynamically incremental fitness function which starts out rewarding for simpler goals, gradually increasing the difficulty of the desired fitness values or goals until a full solutions is found.

2 Related work

A variety of heuristic algorithms have been proposed to solve the partitioning problem. These algorithms include mapping-based methods (Sadayappan and Ercal 1987; Sadayappan, Ercal, and Ramanujam 1990) and clustering-based methods (Kim and Browne 1988; Sarkar 1989; Maheshwari and Shen 1998)

Researchers have also applied GAs to the problem of partitioning parallel programs. Hou et al. (1994) evolve strings which represent processor task schedules. Their approach restricts the actions of genetic operators to ensure the validity of evolved individuals. As a result, some parts of the search space may be unreachable. Correa et al. (1999) improve upon Hou's method to allow the entire search space to be searched. Tsuchiya et al. (1998) propose a GA based scheduler which incorporates the idea of task duplication: one task may be assigned to multiple processors. They compare their GA approach to a heuristic scheduling algorithm called Duplication Scheduling Heuristic (DSH) and show that the GA is able to find better solutions. All of these GA approaches require special methods to ensure the validity of the initial population and to ensure the validity of offspring generated by crossover and mutation. In other words, all individuals generated by these systems must represent "executable" partitions.

3 Algorithm design

We have implemented a novel GA approach for partitioning programs to be executed on parallel systems. We extend the traditional GA (Holland 1975; Goldberg, Korb, and Deb 1989) in two ways. First, we use a flexible representation style which allows the GA to dynamically evolve the structure and value of the solutions. Second, we use an dynamically adaptive, incremental fitness function which initially rewards for simple goals and gradually increases the difficulty of the goals over the generations. Our GA places no restrictions on the individuals that can be formed; both valid and invalid individuals may occur.

```
(3,0)(1,3)(2,2)(1,2)(3,0)(4,3)(5,2)(0,0)(2,1)
```

Figure 1: An example individual.

3.1 Problem representation

Previous work indicates benefits in using location independent problem representations (Goldberg, Korb, and Deb 1989; Holland 1975; Wu and Lindsay 1996) where the information content is not dependent on its location on a GA individual. Such a representation combined with genetic operators that rearrange and exchange information is expected to allow a GA to find building blocks (learn how to arrange related or epistatic information close together). Compactly arranged building blocks (building blocks with low defining length) are expected to be more likely to be transmitted as a whole by the genetic operators during a reproduction event.

Each individual in our GA population consists of a vector of cells. We define a *cell* to be a task and processor pair: (t, p). Each cell indicates that task t is assigned to be processed on processor p. The number of cells in an individual may vary, so individuals in our GA population will vary in length. Figure 1 shows an example individual. The first cell of this individual assigns task 3 to processor 0, the next cell assigns task 1 to processor 3, etc. This representation requires that the number of processors and number of tasks to be performed are known in advance. The problem itself defines the number of tasks to be performed and their dependencies on each other. We assume that the number of available processors is also defined in advance.

The cells on an individual determine which tasks are assigned to which processors. The order in which the cells appear on an individual determines the order in which the tasks will be performed on each processor. Individuals are read from left to right to determine the ordering of tasks on each processor. For example, the individual shown in figure 1 results in the processor assignments and ordering of tasks shown in figure 2. Invalid task orderings will have their fitness value penalized by the fitness function.

The same task may be assigned more than once to different processors. The example individual in figure 1 assigns tasks 1 and 2 twice. Tasks may not be assigned to the same processor more than once. Should a task-processor pair appear more than once on an individual, only the first (leftmost, since individuals are read

Processor 0	Task 3	Task 0	
Processor 1	Task 2		
Processor 2	Task 2	Task 1	Task 5
Processor 3	Task 1	Task 4	

Figure 2: Assignment of tasks from individual in figure 1.

from left to right) pair is active. Any remaining identical pairs are essentially non-coding regions. In the example from figure 1, the second instance of `(3,0)` is not scheduled into the processor lists in figure 2.

As each (task, processor) pair is read from left to right, all active pairs are placed into FIFO queues based on their processor specification. The content of each queue indicates the tasks that will be performed on each processor. The order of the tasks in each queue indicates the order in which the tasks are assigned to be executed on each processor. Thus, the order in which tasks will be performed on each processor depends on the order in which the task-processor pairs appear on an individual.

3.2 Genetic operators

The novel representation requires some modification in the genetic operators used. We use both crossover and mutation in our algorithm. The modified versions of these genetic operators are described here.

3.2.1 Crossover

Recall that each individual consists of a vector of task-processor pairs or cells. Crossover exchanges substrings of cells between two individuals. This allows the GA to explore new solutions while still retaining parts of previously discovered solutions.

Initially we will use random one-point crossover. Random crossover involves two parent individuals. This operator randomly selects a crossover point on each parent and exchanges the segments to the left of the crossover points to form two offspring. Figure 3 shows an example of random crossover.

The crossover rate parameter gives the probability that a pair of parents will undergo crossover. Parents that do not crossover undergo only mutation to form offspring.

3.2.2 Mutation

The mutation rate indicates the probability that a cell will be changed. As a result, the expected number of mutations per individual is equal to the mutation rate multiplied by the length of an individual. If a cell is selected to be mutated, then either the task number or the processor number of that cell will be randomly changed.

3.3 Fitness function

For this problem, a number of factors are expected to contribute to the fitness of an individual ("goodness" of a solution). These factors include but are not limited to:

1. Are all tasks performed? Does an individual contain at least one copy of each task?
2. Are the tasks scheduled on the processors in valid orders?
3. How long will it take the parallel processors to complete all tasks?
4. How well are the tasks distributed among the available processors.

Our task, then, is to determine a fitness function that effectively evaluates and incorporates each of the above components. The current fitness function breaks down the fitness function into two parts. The first part of the fitness function, $task\_fitness$, deals with items one and two above. The second part of the fitness function $processor\_fitness$, deals with items three and four. The actual fitness, $fitness$, of a GA individual is a weighted sum of the above two partial fitness values.

3.3.1 Calculating *task_fitness*

The *task_fitness* component of the fitness function evaluates whether all tasks are represented and in valid order. Because of the complexity of the solutions, we develop an incremental fitness function that changes over time. We will initially start out with a less demanding fitness function, and gradually increase the difficulty of the fitness function over time as the individuals in the GA population improve (Lohn, Haith, Columbano, and Stassinopoulos 1999). As in positive reinforcement training, this strategy rewards for small steps toward the goal, to encourage the algorithm to find the complete goal. We initially reward for finding short valid sequences of tasks. Over time, we increase the length of the sequences that can be rewarded, encouraging the GA to find and maintain longer valid sequences. Eventually the valid sequences will be long enough that the individuals will represent full valid solutions. In addition to "leading the GA towards a goal", this strategy also makes it possible to assign fitness values to valid sub-schedules even if the full represented schedule is not valid. As a result, no special

Randomly select parent 1 crossover point: 2
Randomly select parent 2 crossover point: 4

Parent 1	`(3,0)(1,3)`	`(2,2)(1,2)(4,3)(0,0)(2,1)`
Parent 2	`(3,2)(2,2)(4,1)(2,3)`	`(1,3)(2,2)`

Random crossover produces

Offspring 1	`(3,0)(1,3)`	`(1,3)(2,2)`
Offspring 2	`(3,2)(2,2)(4,1)(2,3)`	`(2,2)(1,2)(4,3)(0,0)(2,1)`

Figure 3: Random one-point crossover randomly selects crossover points on each parent and exchanges the left segments to form offspring.

actions need to be taken to restrict the formation of offspring individuals and the full search space is accessible.

The *task_fitness* component of an individual's fitness is based two main components:

- The percentage of valid sequences of a given length on an individual.
- The percentage of the total number of tasks specified by an individual.

Initially our fitness function will reward for short sequences of valid tasks. A sequence of tasks is valid if the tasks in the sequence are arranged in a valid chronological order. When the average fitness of the GA population exceeds a threshold fitness, the length of the sequence for which the GA searches is increased, thus increasing the difficulty of the fitness function. We anticipate several advantages to this approach. First, previous work indicates that gradually increasing the difficulty of a GA fitness function can result in the formation of more complex solutions (Lohn, Haith, Columbano, and Stassinopoulos 1999). Second, a dynamically changing fitness function is expected reduce the likelihood of premature convergence to partial solutions. A good partial solution (valid subsequence) that does not improve will be worth less over time as the fitness function becomes more demanding.

Calculating raw fitness

The raw fitness of an individual reflects the percentage of sequences of a given length in an individual that are valid sequences. For example, processor 2 in Figure 2 has been assigned three tasks. If our current sequence length is two, processor 2 contains two sequences of length two. Processor 2 contains only one valid sequence of length 2, the sequence `Task1-Task5`. The sequence `Task2-Task1` is not a valid sequence because Task 2 cannot be executed before Task 1.

Assume that the problem to be solved involves P processors and T tasks. Evolution will occur in eras, $era = 0, 1, 2, ..., E-1$. Initially, $era = 0$. The maximum era count, $E \leq T/P$, is a user defined parameter value. The era counter, era, is increased when the average population fitness exceeds a user defined threshold, *thresh* and when the number of individuals with the current maximum possible fitness exceeds a user defined threshold, *thresh_maxfit*.

Let $numtasks(p), p = 0, ..., P-1$, indicate the number of tasks assigned to processor p. To calculate the raw fitness of a processor, we need to consider two things: the first $era+1$ (or fewer) tasks assigned to the processor, and all task sequences of length $era+2$. The first component is important because as era increases, the likelihood of processors containing fewer than $era+2$ tasks increases. We need to reinforce the GA for these shorter sequences in order for them to eventually build up to the measured sequence length. This reinforcement also encourages distribution of tasks among the available processors.

We will first determine the contribution of the first $era+1$ or fewer tasks in a processor. Let

$$subseq(p) = \begin{cases} 1 & \text{if } numtasks(p) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

and let

$$valseq(p) = \begin{cases} 1 & \text{if the first } era+1 \text{ or fewer} \\ & \text{tasks in processor } p \text{ are in} \\ & \text{valid order} \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Equations 1 and 2 refer to individual processors. To calculate the contribution over all processors (the contribution for the entire individual), we let

$$Subseq = \sum_{p=1}^{P-1} subseq(p)$$

$$Valseq = \sum_{p=1}^{P-1} valseq(p).$$

We will next determine the contribution of all sequences of length $era + 2$ in a processor. Let

$$s(p) = \#\text{ sequences of length } era + 2 \text{ in processor } p \quad (3)$$

and let

$$v(p) = \#\text{ valid seq. of length } era + 2 \text{ in proc. } p. \quad (4)$$

Combining equations 3 and 4 to determine the contribution over all processors we let

$$S = \sum_{p=1}^{P-1} s(p) \qquad V = \sum_{p=1}^{P-1} v(p).$$

The $raw\_fitness$ for an individual is then calculated with the following equation

$$raw\_fitness = \frac{Valseq + V}{Subseq + S}. \quad (5)$$

Calculating the task ratio

In addition to encouraging the system to find valid sequences of tasks, we also want to encourage the system to include at least one copy of each task in each solution. We define the *task_ratio* to be the percentage of distinct tasks from the total tasks in the problem that are represented in an individual. The *task_ratio* is calculated with the following equation:

$$task\_ratio = \frac{\begin{array}{c}\textit{number of distinct tasks}\\ \textit{specified on an individual}\end{array}}{\begin{array}{c}\textit{total number of tasks}\\ \textit{in the problem.}\end{array}} \quad (6)$$

This factor penalizes solutions that do not contain at least one copy of every task. Once all tasks are represented in an individual, this penalty becomes null.

Effective fitness

The effective $task\_fitness$ of an individual is the product of equations 5 and 6.

$$task\_fitness = raw\_fitness * task\_ratio \quad (7)$$

This value makes up the first component of the fitness of a GA individual.

3.3.2 Calculating *processor_fitness*

To fully optimize the use of parallel processors, the GA must be able to distribute tasks among the available processors. The $processor\_fitness$ component of the fitness function addresses this issue.

Suppose t is the run time for a solution represented by an individual. Let $serial\_len$ equal the length of time required to complete all tasks serially on a single processor and let $super\_serial\_len = P * serial\_len$ where P equals the number of processors. Any reasonable solution should give $t \ll super\_serial\_len$, making $super\_serial\_len$ a safe but reasonable upper bound to solution run time. The goal of the GA is to minimize t. The $processor\_fitness$ first calculates the difference between $super\_serial\_len$ and t then calculates what proportion of $super\_serial\_len$ this difference represents:

$$processor\_fitness = \frac{super\_serial\_len - t}{super\_serial\_len}.$$

As a result, $processor\_fitness$ is inversely proportional to t. As the run time of a solution decreases, the amount that $processor\_fitness$ contributes to the individual's full fitness increases.

It is important to note that although the theoretical maximum value of *processor_fitness* is 1.0, in practice, this value can not be achieved. For *processor_fitness* to equal 1.0, the run time, *t*, of a solution would have to be zero. Since all tasks obviously require non-zero run time, *t* will never be zero for valid individuals.

3.3.3 Calculating *fitness*

The full fitness of an individual is a weighted sum of the $task\_fitness$ and $processor\_fitness$:

$$fitness = (1-b)*task\_fitness + b*processor\_fitness.$$

Our initial experiments use $b = 0.1$. The value of b may be adjusted in future runs.

Even though the theoretical maximum value of *fitness* is 1.0, this value will never be achieved in practice because the *processor_fitness* component of the fitness function can never reach 1.0 in practice.

The value of $fitness$ is returned to the GA by the fitness function as the fitness of an individual.

4 Experimental details

A parallel program can be represented by its data flow graph $G = (V, E)$. This graph is a node-labeled and edge-labeled directed acyclic precedence graph (APG), where V is a set of nodes that represent sequential processes and E is a set of directed edges that specify both precedence constraints and communication paths

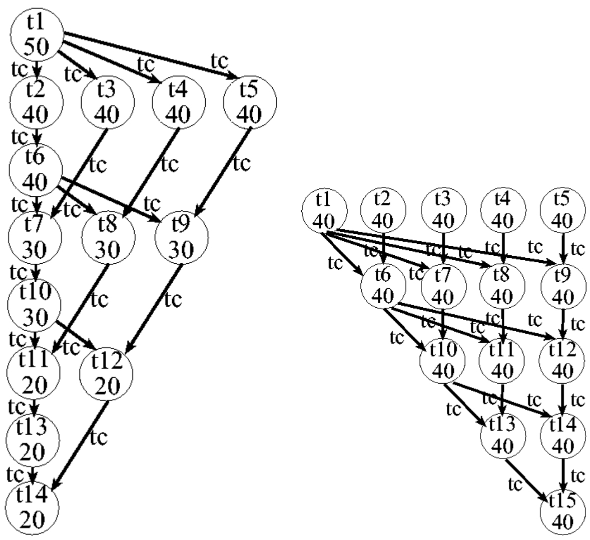

Figure 4: Description of task dependencies for problems C and B in (Tsuchiya, Osada, and Kikuno 1998).

among nodes. In an APG, each node gives the execution cost time and each edge represents the communication cost time required to pass data from one node to another during program execution.

We test our GA on several problems described by (Tsuchiya, Osada, and Kikuno 1998), specifically problems B and C from their paper. The APG for these problems are shown in figure 4. The communication time, tc, is set to 25 for problem B and 20 for problem C. For problem B, Tsuchiya et al. (1998) obtained a minimum run time of 300 with their GA and 315 with DSH. For problem C, they obtained a minimum run time of 260 with their GA (Tsuchiya 2001) and 260 with DSH.

The details of our GA are described in section 3. We used the following parameter settings in our experiments:

Population size	=	200
Crossover type	=	Random 1-point
Crossover rate	=	1.0
Mutation rate	=	0.005
Selection method	=	Tournament
Stopping condition	=	1500 generations.

Our GA performs 300,000 evaluations (= population size 200 × 1500 generations) in each run, making it comparable to the data from (Tsuchiya, Osada, and Kikuno 1998) (population size 30 × 10,000 generations = 300,000 evaluations). On average, however, our GA finds the best solution in about 600 generations, so significantly fewer evaluations are actually needed. We

Minimum run time

	$thresh = 0.7$	$thresh = 0.75$
$b = 0.1$	303 (6.32)	301.5 (4.47)
$b = 0.2$	304.5 (10.12)	302 (4.83)

Maximum fitness

	$thresh = 0.7$	$thresh = 0.75$
$b = 0.1$	0.9874 (0.0003)	0.9874 (0.0002)
$b = 0.2$	0.9746 (0.0008)	0.9748 (0.0003)

Figure 5: Results for problem B: average (and standard deviation) over ten runs.

tested the following variations in the fitness function parameters:

$$\begin{aligned} thresh &= 0.7, 0.75 \\ b &= 0.1, 0.2 \\ thresh\_maxfit &= 0.1\ . \end{aligned}$$

For each experiment, we also include a set of runs in which the fitness function simply rewards for valid runs with minimal run time and does not use an incremental schedule. These runs are used as a comparison to judge the merits of our GA using an incremental fitness function.

5 Results

Our initial experiments test two values of *thresh* (which determines when the *era* counter will advance) and two values of b (which determines the contribution of the two main components of the fitness function). Ten runs were executed for each experiment. We report here the average performance over each set of ten runs.

Figure 5 gives the results for problem B. The top table in figure 5 shows the run time of the best solution (solution with minimum run time) found, averaged over ten runs. The minimum run time found is 300. This minimum value is achieved in 80% of the 40 runs represented in figure 5 and is comparable to the best GA solution found in (Tsuchiya, Osada, and Kikuno 1998). The bottom table in figure 5 shows the average maximum fitness found in the runs. While there is not a large difference in the average run times of each set of experiments, the data suggest that a lower value of b appears to produce solutions with higher fitness.

In addition to these experiments on problem B, we also run an experiment consisting of a set of ten baseline runs in which the fitness function simply rewarded for minimum valid run times. No incremental fitness is given in these runs. Seven of the ten runs are unable

Minimum run time

	$thresh = 0.7$	$thresh = 0.75$
$b = 0.1$	267 (9.49)	274 (13.50)
$b = 0.2$	275 (15.81)	271 (9.94)

Maximum fitness

	$thresh = 0.7$	$thresh = 0.75$
$b = 0.1$	0.9848 (0.0005)	0.9844 (0.0007)
$b = 0.2$	0.9687 (0.0018)	0.9692 (0.0011)

Figure 6: Results for problem C: average (and standard deviation) over ten runs.

to find any valid partitions within the 500 generations. Three find solutions with run times of 500, 435, and 655, all significantly higher than the run times found using the incremental fitness function. Closer inspection of these runs reveals that these solutions are not actually evolved. Rather they are randomly generated individuals that just happen to be valid individuals. They appear for only for a single generation and do not produce any viable offspring. In fact, all three of these solutions were randomly generated in the initial population and then immediately destroyed. No other valid solutions were found in any of these runs.

Figure 6 gives the results for problem C. The top table in figure 6 shows the run time of the best solution (solution with minimum run time) found, averaged over ten runs. The minimum run time found is 260, occurring in almost half of the 40 runs reported above. This value is comparable to the best GA and DSH solutions from (Tsuchiya 2001; Tsuchiya, Osada, and Kikuno 1998). The bottom table in figure 6 shows the average maximum fitness found in the runs. Again, a lower value of b appears to produce solutions with higher fitness. We also run a baseline experiment for problem C consisting of ten runs without incremental fitness. Eight of the ten runs are unable to find a valid solution within 500 generations. Two runs find solutions with a run time of 590 and 380. These solutions, again, appear to be random occurrences, the only valid individuals generated in these runs.

Figure 7 shows an example of how the population fitness of a run evolves in our GA. The top line shows the best population fitness at each generation. The bottom line shows the average population fitness at each generation. The vertical lines indicate the generations at which the *era* counter is incremented. The start of each era is indicated at the top of the graph. The average population fitness climbs within each era. Each time the *era* counter is incremented, however, the difficulty level of the fitness function increases and the

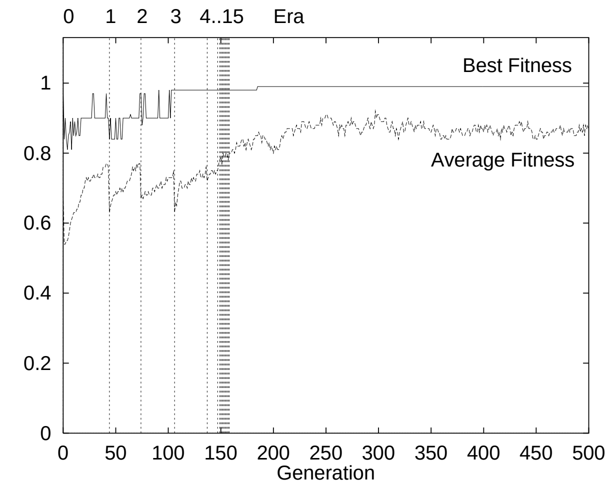

Figure 7: Evolution of population fitness in response to increasing eras.

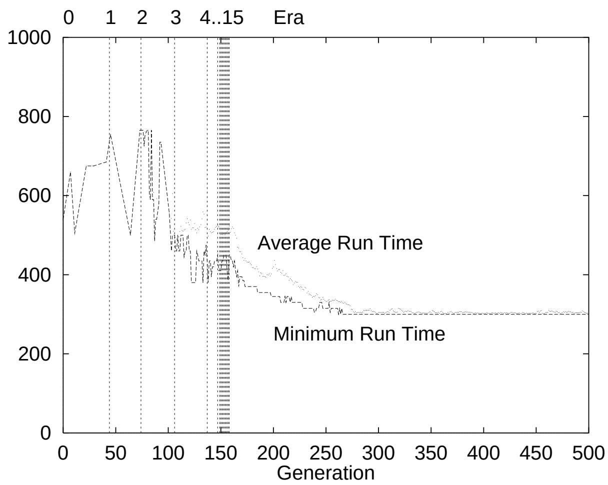

Figure 8: Evolution of run time in response to increasing eras.

average fitness of the population drops. After about six eras in this run, there are apparently enough valid task sequences to allow the remaining eras to increment once per generation until the maximum $era = 15$ is reached. Figure 8 shows the evolution of run time in the same run. The run times of the evolved solutions seem to be less reactive to the change in eras.

6 Conclusions and future work

We introduce a novel GA approach to evolving schedules that partition the tasks of programs to be executed on parallel processors. Our GA successfully solves this problem, finding solutions that are comparable or better than solutions found by previous methods. The key advantage of our GA over previous GA approaches to this problem is that our GA does not use special operators or repair mechanisms to restrict the search to only valid regions of the search space. As a result, we eliminate biases that may occur due to the special operators, we know that all individuals within the search space are reachable, and we eliminate the need to re-tune or revise the special operators and repair mechanisms each time a new problem is introduced to be solved.

Two unique features distinguish our GA from traditional GA programs. First, this GA uses a dynamically incremental fitness function which starts out rewarding for simpler goals, gradually increasing the difficulty of the desired fitness values or goals until a full solutions is found. Second, this GA uses a flexible representation style which allows the GA itself more control over both the structure and the value of the evolved solutions.

It is clear from our baseline experiments that this is a difficult problem for a GA to solve if there are no restrictions placed on the evolutionary process. We modify the problem specific components of a GA – the problem representation and the fitness function – to encourage the GA to create and maintain valid individuals. The incremental fitness function initially rewards the GA for finding short valid sequences of tasks. As fitness improves, the lengths of these sequences increase, thus rewarding the GA for finding and keeping longer valid sequences. Valid individuals are further evaluated with respect to their quality so that shorter run times are rewarded.

We hope to further improve the performance of our GA with future developments. Issues that we plan to address include an analysis of the effectiveness of our representation format, an examination of the use of alternate genetic operators, and a study of the effects of variable genome length in a GA.

Acknowledgements

This research is supported in part by the Air Force Research Laboratories and Science Applications International Corporation. The authors would like to thank the anonymous reviewers for their helpful comments.

References

Bokhari, S. H. (1981). On the mapping problem. *IEEE Trans. Comput. 30*(3), 207–214.

Correa, R. C., A. Ferreira, and P. Rebreyend (1999). Scheduling multiprocessor tasks with genetic algorithms. *IEEE Trans. Parallel and Distrib. Syst. 10*(8), 825–837.

Goldberg, D. E., B. Korb, and K. Deb (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems 3*, 493–530.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

Hou, E. S., N. Ansari, and H. Ren (1994). A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel and Distrib. Syst. 5*(2), 113–120.

Kim, S. J. and J. C. Browne (1988). A general approach to mapping of parallel computations upon multiprocessor architectures. In *Proc. Int. Conf. Parallel Processing*, pp. 1–8.

Lohn, J. D., G. L. Haith, S. P. Columbano, and D. Stassinopoulos (1999). A comparison of dynamic fitness schedules for evolutionary design of amplifiers. In *Proc. 1st NASA/DoD Workshop of Evolvable Hardware*, pp. 87–92.

Maheshwari, P. and H. Shen (1998). An efficient clustering algorithm for partitioning parallel programs. *Parallel Computing 24*, 893–909.

Sadayappan, P. and F. Ercal (1987). Nearest-neighbor mapping of finite element graphs onto processor meshes. *IEEE Trans. Comput. 36*(12), 1408–1424.

Sadayappan, P., F. Ercal, and J. Ramanujam (1990). Cluster partitioning approaches to mapping parallel programs onto a hypercube. *Parallel Computing 13*, 1–16.

Sarkar, V. (1989). *Partitioning and scheduling parallel programs for multiprocessors*. MIT Press.

Tsuchiya, T. (2001). Personal communication: updated results.

Tsuchiya, T., T. Osada, and T. Kikuno (1998). Genetic-based multiprocessor scheduling using task duplication. *Microprocessors and Microsystems 22*, 197–207.

Wu, A. S. and R. K. Lindsay (1996). A comparison of the fixed and floating building block representation in the genetic algorithm. *Evolutionary Computation 4*(2), 169–193.