Improved Multiprocessor Task Scheduling Using Genetic Algorithms

Michael Bohler
Sensors Directorate
Air Force Research Laboratory
WPAFB, OH 45433
Michael.Bohler@sensors.wpafb.af.mil

Frank Moore
Systems Analysis Dept.
Miami University, 230 L Kreger Hall
Oxford, OH 45056
moorefw@muohio.edu

Yi Pan
Computer Science Dept.
University of Dayton, 300 College Park
Dayton, OH 45469-2160
pan@udcps.cps.udayton.edu

Abstract

Efficient multiprocessor task scheduling is a long-studied and difficult problem that continues to be a topic of considerable research. This NP-complete problem is typically solved using a combination of search techniques and heuristics. Traditional solutions require a deterministic search of the solution space, which is computationally and temporally exhaustive. Genetic algorithms are known to provide robust, stochastic solutions for numerous optimization problems. This paper describes the design and implementation of a genetic algorithm for minimizing the schedule length for a general task graph to be executed on a multiprocessor system. The implementation is scalable and adaptable to a variety of task graphs and parallel processing systems. Several improvements over state-of-the-art approaches lead to a vigorous solution.

1. Introduction

The multiprocessor scheduling problem is generally stated this way: given a multiprocessor computing system and numerous tasks to execute, how does one efficiently schedule the tasks to make optimal use of the computing resources? In general, a deterministic search of the solution space to identify an optimal solution to this NP-complete problem is computationally and temporally exhaustive. The problem difficulty depends chiefly upon the following factors: the number of tasks, execution time of the tasks, precedence of the tasks, topology of the representative task graph, number of processors and their uniformity, inter-task communication, and performance criteria. Classic solutions to this problem (De Jong and Spears 1989; Ercal 1988; Zahorjan and McCann 1990) use a combination of search techniques and heuristics. While these techniques often produce adequate solutions, the resulting schedule is usually suboptimal. These techniques are also criticized for lacking both scalability and performance guarantees.

Genetic algorithms (Holland 1975; Goldberg 1989) provide robust, stochastic solutions for numerous optimization problems. This paper describes the implementation of a genetic algorithm for minimizing the schedule length of a task graph to be executed on a multiprocessor system, and identifies several improvements over state-of-the-art solutions (Hou, Ansari and Ren 1994; Kwok and Ahmad 1997).

In this paper, the multiprocessor scheduling problem is given as a parallel program represented by an acyclic directed task graph. Precedence relations are derived from the task graph, and the execution time of each task is randomly set. All processors are assumed to be identical, are non-preemptive, and use the shared memory model (zero communication delays between tasks). Unlike previous solutions, the program developed in this project may be applied to a wide range of scheduling problems; it is scalable and adaptable to a variety of task graphs and parallel processing systems. The number of tasks is easily changed to allow schedule optimization for a variety of task graphs with various numbers of tasks. Task and task graph characteristics such as execution times and precedence are readily modified. The number of processors on the target multiprocessor system is freely modifiable to yield schedules for arbitrary parallel processors. Other parameters that are easily modified include the number of iterations of the genetic algorithm (number of generations) and population size. Finally, the program described herein has been constructed to execute efficiently on a personal computer, eliminating the need for high performance workstations traditionally employed in this application.

2. Related Work

Hou, Ansari, and Ren (Hou, Ansari and Ren 1994) presented a genetic algorithm that solved the multiprocessor scheduling problem for a directed task graph using simple genetic operations in combination with a variety of random selection techniques. This algorithm provided a basis for the improved genetic task scheduler implemented for this research.

3. Implementation

Throughout this description, references to the actual C++ implementation are contained in *italics*. The executable program generated by this implementation is named *SCHEDULE*. Following traditional genetic algorithm methodology (Holland 1975; Davis 1991), candidate solutions are evolved by *SCHEDULE* from the random initial population using fitness-proportionate reproduction, crossover, and mutation.

3.1 Tools and Language

SCHEDULE was implemented in the C++ programming language using the Borland Turbo C++ v. 4.5 compiler. Extensive use was made of the Array and Set classes of the Borland Container Class Library included in the compiler. Frequent use was made also of the Borland random number generator. The computer used for this development was a Dell Lattitude XPi laptop computer with an Intel Pentium 150 MHz processor.

3.2 Chromosome Representation

SCHEDULE manipulates chromosomes that contain a sequence of task identifiers whose order indicates the order of execution of tasks assigned to a processor. Multiple strings collectively containing all tasks to be executed on the processors in a parallel computing system constitute a schedule and are called *genomes* (Figure 1). Note that the precedence relations of the tasks in each string are strictly enforced. In this manner, the order of execution of tasks represented in a string is legal. This representation also allows us to ignore interprocessor precedence – the order between tasks on different processors – during genome-manipulating operations; interprocessor precedence is considered only when the finish time of the schedule is calculated. The fitness of a particular genome is inversely proportional to the finish time of the corresponding schedule: smaller finish times indicate better fitness.

3.3 Initialization

SCHEDULE begins in *main()* by reading two input files describing the task graph. The first file contains task numbers and their execution times. A master list of task objects, *master*, is constructed in *build_master()* from the information in this file. Tasks are numbered in consecutive increasing order beginning with 1. *Master* is implemented as an array of sets whose indices correspond to the task number and where each set contains one task object.

The second input file contains a list of task pairs that represent the edges leading from one task to the next. This edge data conveys information about task precedence. Edge data allows *build_sets()* to construct of two arrays of sets, *pred_array* and *succ_array*, whose indices correspond to task numbers and point to sets of tasks which either precede or succeed that particular task.

Additional information derived from the edge data is task height. Height is calculated and assigned to each task as a means of ordering tasks on a processor. The height of a task is zero for any task with no predecessors; otherwise, height is defined as one plus the maximum height from the set of all immediate predecessor tasks. By examining *pred_array*, the height of each task is calculated in *compute_height()* and stored with each task object in *master*.

The "height prime" (h') of each task is derived next. Hou *et al* (Hou, Ansari and Ren 1994) introduced h' to allow variability in task ordering and thus increase the number of schedules in the solution space. The value of h' is defined as a random integer ranging from the height of the task itself to the minimum height of all immediate successors, minus one. The example in Figure 2 illustrates how task 4 can have a h' value of either two or three. The height of task 4 is two and the minimum height from the set of

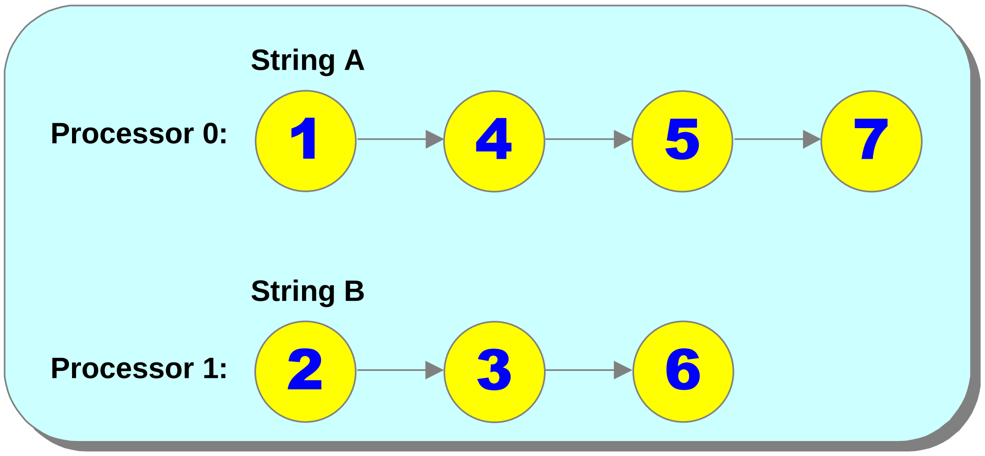

Figure 1. An Example Genome - 7 tasks scheduled on a dual processor system

immediate successor tasks is four. Therefore the h' value of task 4 can be either two or three. When a particular schedule is created, one of the two values is randomly selected as the h' value of the task. Task 4 can be ordered on a processor after tasks of height 1 and before tasks of height 4 and not violate ordering constraints. This example illustrates how h' adds variability to task ordering and increases the number of schedules for evaluation. (Kwok and Ahmad 1997) achieved comparable results using the start-time minimization method.

The function *compute_height_prime()* uses *succ_array* to randomly determine a h' value for each task and stores the result with each task object within *master*. At this point, *succ_array* is no longer needed and is deleted with the function *delete_set()*. A graph-generating program described later in this report creates input files containing task graph information data.

Another data structure created during initialization is the *at_height* array. *At_height* is an array of sets whose indices correspond to h' values, where each set contains all the tasks with that particular h' value. This array is useful for quickly finding all the tasks with the same h' value, as required by the mutation operator described in Section 3.5.2.

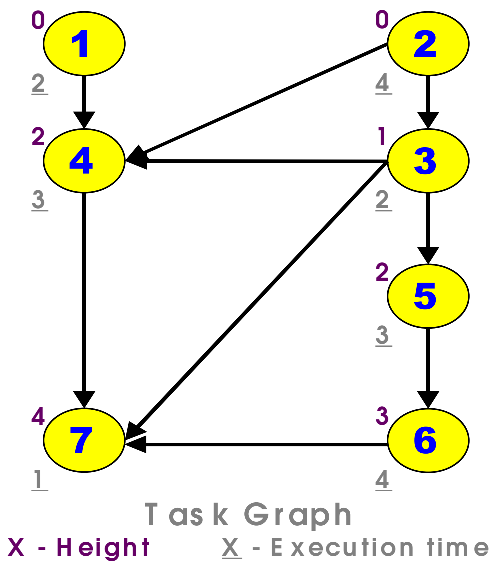

Figure 2. Task Graph for height' Example.

3.4 Generating the Initial Schedules

Generate_schedule() generates the initial schedules. The number of schedules to be generated is determined by the global variable defined in defines.h, *POP*, which is the population size or number of genomes in each generation. The schedules are stored in a three-dimensional array of sets named *genomes*. The first dimension of *genomes* distinguishes one genome from the next. The second dimension identifies the processor and the third dimension demarcates the tasks. Due to its potentially large size, *genomes* is declared globally to avoid overflowing the run-time stack.

The process for selecting tasks and ordering them on p processors provided an opportunity to eliminate a shortcoming in the algorithm described by Hou *et al* (Hou, Ansari and Ren 1994). In their description, the tasks are partitioned into sets G(h), which are sets of tasks with height h, according to their value of h'. For each of the first p-1 processors, a random number r is generated in the range 0 to N for each G(h), where N is the number of tasks in the set. r determines the number of tasks selected from the set G(h) and assigned to the current processor p. This approach was implemented but frequently caused either very many or very few tasks from a set to be assigned to a single processor, resulting in generally poor initial schedules. Also, their description states that they "pick r tasks from G(h), remove them from G(h), and assign them to the current processor." It is not specified whether the first r tasks in the set are always picked, or whether the tasks are selected in some random fashion.

The approach implemented by Hou *et al* (Hou, Ansari and Ren 1994) is valid for the limited case of two-processor systems, but lacks scalability in the number of processors used and is unacceptable for 3 or more processor systems. Their task selection strategy has the effect of, on average, assigning ½ of the tasks to the first processor, then ½ of the remaining tasks to the next processor, and then ½ of the remaining tasks to the next processor, and so on. As a result, in many cases, some processors are assigned many tasks while others are assigned very few or none. The potential for maximum parallelization of the tasks is lost.

For each task in a set G(h) and each processor p, *SCHEDULE* generates a random floating-point number r between zero and one and the task is assigned to p if $r < 1/n$, where n is the number of processors in the system. This technique tends to distribute the tasks more evenly over all of the available processors and maximizes the potential to exploit parallelism.

3.5 The Genetic Algorithm

SCHEDULE terminates after *MAXGEN* generations have been evolved, or when a suitable solution has been produced. The actual number of generations needed depends upon the number of tasks and population size. Generally, the fewer the number of tasks and the larger the population size, the fewer the number of generations required for a convergent solution. Continuing to run the algorithm for more generations does not improve the result.

Crossover is the first genetic operation performed on the genomes. In each generation, all the genomes in the population are randomly paired for crossover. For each genome pair, *crossover()* is called with a probability of *PROBCROSS*. The crossover site, or location where crossover occurs in the genome pair, is randomly selected from the valid crossover sites in a genome pair. A valid crossover site is a point in both genomes where the value of h' for all the tasks before the site is lower than the value of h' for all the tasks after the site. A valid site must also have at least one task both before and after the site. Otherwise, either none or all tasks would be swapped, and the resultant genome would be no different than the original.

Following the crossover operation, mutation is performed by calling *mutate()* with probability *PROBMUTATE* for each genome. When a genome is picked to be mutated, a randomly selected task is swapped with another randomly selected task with the same h', both within the same genome.

After breeding a new generation of schedules, the fitness of each genome in the population is computed by calling *fitness()*. In *fitness()*, the finish time of each genome is first computed and stored in the array *finish_times[]*. Finish times are determined by constructing a Gantt chart for the schedule represented in the genome. At this point, interprocessor task precedence is important and must be taken into consideration. Figure 3 illustrates the Gantt chart for the task graph in Figure 2 and the schedule of Figure 1. Since tasks within a chromosome are ordered by h', no precedence constraints are violated. Task precedence between processors must be enforced to when considering the time to complete all of the tasks scheduled in a genome. Notice, for instance, that task 4 cannot execute until task 3 has finished and task 6 cannot begin until task 5 has completed. Also notice that tasks 1 and 2 are not dependent on each other and can execute concurrently. When the finish times for all the genomes in a population are known, the fitness of the genomes can be calculated. Since the objective is to minimize the finishing time of the schedule, the fitness value of the schedule will be the maximum finishing time observed in a population, minus the finish time of the schedule (*fitval[] = worst_time - finish_time[]*). One is added to the fitness value of all genomes so that even the genome with the worst fitness value is non-zero, giving that genome a chance to be selected for reproduction. Fitness values are returned to the main function in *fitness_values[]*.

Reproduction is applied by calling *reproduce()*. *Reproduce()* creates a new population of genomes by selecting genomes from the current population using weighted random numbers based on their fitness values. Thus, genomes with higher fitness values (shorter finish times) have a better chance of surviving to the next generation.

It is possible for two genomes comprised of totally different schedules to have equal fitness values. *SCHEDULE* guarantees that one copy of a genome with the best overall fitness value survives to the next generation. The particular genome selected is the first one in *genomes* with the best fitness value in the current generation. The rest of the genomes to be reproduced are selected via a biased roulette wheel weighted on genome fitness values. Conceptually, a roulette wheel is constructed where each genome occupies a slot on the wheel whose area is proportional to the fitness value of the genome. Random numbers are generated and used to index into the wheel and select a genome to be passed to the next generation. Because genomes with higher fitness values occupy more space on the wheel, they are more likely to reproduce to the next generation.

Consideration was given to propagating all genomes with the best fitness value to the next generation. However, it was found that this approach quickly removed variability from the population and frequently caused rapid convergence to very poor schedules.

3.6 Schedule Selection

At the end of a run, a call to *fitness()* returns the fitness values, the best finish value is found, and the first genome of those with the best finish time is selected for the result schedule. The output file *out.txt* is generated during the course of the program. Examining this file reveals the desired schedule.

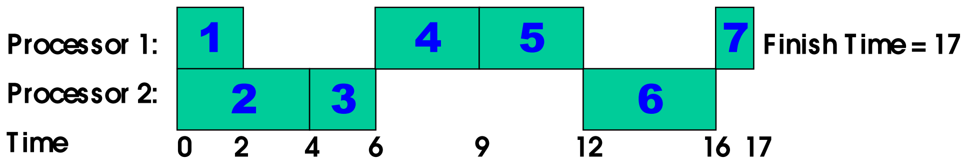

Figure 3. A Gantt chart displays a schedule for 7 tasks on a dual processor system

3.7 *SCHEDULE* Algorithm

The algorithm for the *SCHEDULE* program is given here:

Schedule (S)

```
Solve the multiprocessor scheduling problem:
S1. Initialize: Build the master task list. Build the
    successor and predecessor task lists.
S2. Compute h' for every task.
S3. Call Generate-Schedule and store the created
    genomes (schedules) in GENOMES.
S4. Repeat S5-S6 GENERATIONS number of
    times.
S5. Randomly pair the genomes in GENOMES and
    call Crossover for each pair with a probability of
    PROBCROSS.
S6. For each genome in GENOMES, call Mutate
    with a probability PROBMUTATE
S7. Compute the fitness value of each genome. Store
    the results in FITNESS_VALUES.
S8. Call Reproduce.
S9. Compute the fitness value of each genome. Store
    the results in FITNESS_VALUES. Select the
    genome (schedule) from GENOMES with the
    best (smallest) fitness value.
```

Generate-Schedule (GS)

```
Randomly generate initial schedules for the given
    task graph for a multiprocessor system with p
    processors:
GS1. Build N genomes (schedules) where N is the
    population size. For each genome in the
    population, repeat GS2-GS5.
GS2. Partition the tasks into different sets G(h)
    according to the value of h'.
GS3. For each of the first p-1 processors do step
    GS4.
GS4. For each set G(h) do step GS5.
GS5. For each task in the set, randomly generate a
    number r between 0 and 1. If $r < 1/p$, select the
    task from G(h) and assign it to the current
    processor.
GS6. Assign the remaining tasks in the sets to the
    last processor.
```

Crossover (C)

```
Perform the crossover operation on two genome
    pairs A and B:
C1. Select crossover sites: Generate a random
    number c between 0 and the maximum h' of the
    task graph.
C2. Do C3 for each processor $P_i$ in genomes A and
    B.
C3. Find crossover sites: Find the task $T_{ji}$ in
    processor $P_i$ that has height c, and $T_{ki}$ is the task
    following $T_{ji}$ where $c = h'(T_{ji}) < h'(T_{ki})$ and
    $h'(T_{ji})$ are the same for all i.
C4. Do C5 for each processor $P_i$ in genome A and B.
C5. Using the crossover sites selected in C3,
    exchange the bottom parts of genomes A and B.
```

Reproduce (R)

```
Perform the reproduction operation on the
    population of genomes POP and generate a new
    population NEWPOP:
R1. Construct roulette wheel: Let FITNESS_SUM
    be the sum of all fitness values of genomes in
    POP; form FITNESS_SUM slots and assign
    genomes to occupy the number of slots according
    to the fitness value of the genome.
R2. Select the first genome in POP with the highest
    fitness value. Add this genome to NEWPOP.
R3. Let NPOP be the number of genomes in POP.
    Repeat R4 NPOP-1 times.
R4. Generate a random number between 1 and
    FITNESS_SUM. Use it to index into the slots to
    find the corresponding genome. Add this genome
    to NEWPOP.
```

Mutate (M)

```
Mutate a genome to form a new genome:
M1. Randomly pick a task $T_i$ that has at least one
    other task with the same h'.
M2. Randomly select another task $T_j$ with same h' as
    $T_i$.
M3. Form a new genome by exchanging the two
    tasks $T_i$ and $T_j$ in the genome.
```

There are several differences between *SCHEDULE* and the algorithm presented by Hou *et al* (Hou, Ansari and Ren 1994). As described in Section 3.4, the most important of these is the method by which tasks are allocated to processors in GS4 of the Hou *et al* algorithm and GS5 in the *SCHEDULE* algorithm. For each task in a set G(h), *SCHEDULE* generates a random floating-point number r between zero and one and the task is selected if $r < 1/n$, where n is the number of processors in the system. This technique offers an improvement over the approach by Hou *et al* as it spreads the tasks more evenly over all the available processors and maximizes the potential to exploit parallelism when $n > 2$.

The goal of *SCHEDULE* is to find the shortest execution time for a particular set of tasks executing on a specified multiprocessor system. In this implementation, an optimal schedule for a particular task graph may depend on one or more tasks being assigned the right h' value, assigned to the right processor, and placed in the right sequence. These are all probabilistic events and it is evident that the larger the population, the greater the chance that the right combination of events will occur. However, the h' values assigned to tasks are constant throughout the execution of the program. If the optimal solution depends on the correct value of h' being assigned to one or more tasks, and if that doesn't occur, then the optimal solution for that run will never be found. In this particular case, increasing the population size will not benefit the chance of finding the optimal solution.

4. Results

The genetic algorithm was implemented and tested on arbitrary task graphs of 20, 40 and 50 tasks. A program was written to randomly generate the data to represent task graphs of arbitrary size. The output of this program was used as the input data for the *SCHEDULE* program. The task data generation program was written to arbitrarily allow each task to have as many as four immediate successors. Each task was allowed a random execution time of 1 to 40 time units. Figure 4 shows the task graph used for the test case of 40 tasks.

Table 1 shows the results (*best finish time found and **average of 20 runs) of schedules for task graphs with the number of tasks indicated. All results are based on schedules for a four-processor system. *SCHEDULE* took approximately 2 seconds to complete for 20 tasks, 14 seconds for 40 tasks, and 60 seconds for 50 tasks. Execution time depends primarily on the number of generations.

Suboptimal solutions identified by *SCHEDULE* were often the result of premature convergence (Schaffer, Eshelman and Offutt 1991). This condition was evidenced by many genomes arriving at the same suboptimal schedule after relatively few generations. Various strategies have been proposed to avoid premature convergence, including employing mating strategies, managing crossover, and managing the population (Eshelman and Schaffer 1991). *SCHEDULE* could benefit from one or more of these techniques.

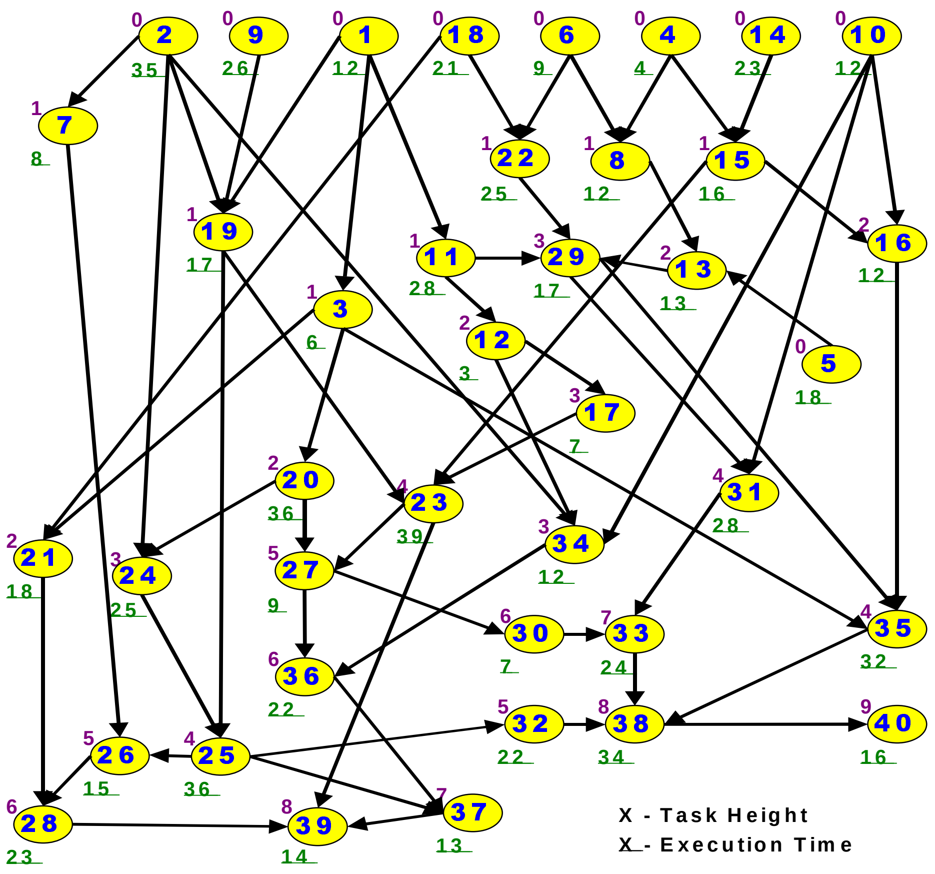

Figure 4. Test Task Graph with 40 Tasks

Tasks	Generations	Population	Prob. Crossover	Prob. Mutate	Optimal Schedule	*GA	**GA	(GA – Opt)/ Opt
20	20	10	1.0	0.1	159	159	161	1.2 %
40	100	26	1.0	0.1	218	230	243	11.4 %
40	100	26	1.0	0.2	218	218	233	6.8 %
50	100	18	1.0	0.2	unknown	319	339	unknown
50	500	18	1.0	0.2	unknown	319	331	unknown

Table 1. Test Results (*best finish time found and **average over 20 runs)

5. Conclusions

In this research, a genetic algorithm was implemented in the *SCHEDULE* program to solve the multiprocessor scheduling problem. Created to run on a personal computer, *SCHEDULE* is a simple and easy-to-use tool for modeling and scheduling tasks on a multiprocessor system. It is easily modified to correctly schedule arbitrarily complex task graphs onto specified multiprocessor architectures. The user can change the number of processors in the target system, the task graph representation (the number of tasks and their execution times), and the parameters of execution (the number of generations and population size). Minor changes to the program would easily support the introduction of interprocessor communication delays and overhead costs of the system (Jesik *et al* 1997), as well as other options.

In only 20 generations, the genetic algorithm found an optimal solution in 75% of the runs for 20 tasks, and was within 1.2% of the optimal solution on average. For the larger problem of 40 tasks the genetic algorithm found a good solution in 100 generations. A suitable solution for 50 tasks was found in 500 generations. Limitations of the compiler caused the effectiveness of the solution for larger problems to degrade. In future research, the adaptable and scalable methodology established by this investigation may be applied to increasingly complex task graphs.

6. References

Davis, L. 1991. *Handbook of Genetic Algorithms*, Van Nostrand Reinhold.

De Jong, K. A. and W. M. Spears 1989. “Using Genetic Algorithms to Solve NP-complete Problems”, in *Proceedings of the 3rd International Conference on Genetic Algorithms*, pp. 124-132, Morgan Kaufmann.

Ercal, F. 1988. “Heuristic Approaches to Task Allocation for Parallel Computing”, Ph.D. Thesis, Ohio State University.

Eshelman, L. J. and J. D. Schaffer 1991. “Preventing Premature Convergence in Genetic Algorithms by Preventing Incest”, in *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 115-122, Morgan Kaufmann.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley.

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

Hou, E. S. H., N. Ansari and H. Ren 1994. “A Genetic Algorithm for Multiprocessor Scheduling”, in *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5 No. 2, pp. 113-120, Feb. 1994.

Jesik, G., R. Kostelac, I. Lovrek and V. Sinkovic 1998. “Genetic Algorithms for Scheduling Tasks with Non-negligible Intertask Communication onto Multiprocessors”, in Koza, J. R. *et al*, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, July 22-25, 1998, University of Wisconsin-Madison, p. 518, Morgan Kaufmann.

Kwok, Y.-K. and I. Ahmad 1997. “Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using A Parallel Genetic Algorithm”, in *Journal of Parallel and Distributed Computing*, Vol. 47, No. 11, pp. 58-77, Nov. 1997.

Schaffer, J. D., L. J. Eshelman and D. Offutt 1991. “Spurious Correlations and Premature Convergence in Genetic Algorithms”, in *Foundations of Genetic Algorithms*, pp. 102-114, Morgan Kaufmann.

Zahorjan, J. and C. McCann 1990. “Processor Scheduling in Shared Memory Multiprocessors”, in *Performance Evaluation Review*, 18(1), pp. 214-225, May 1990.