Comparison Between Synchronous and Asynchronous Implementation of Parallel Genetic Programming

Shisanu Tongchim
Department of Computer Engineering
Chulalongkorn University
Bangkok 10330, Thailand
g41stc@cp.eng.chula.ac.th

Prabhas Chongstitvatana
Department of Computer Engineering
Chulalongkorn University
Bangkok 10330, Thailand
prabhas@chula.ac.th

Abstract

An evolutionary method such as Genetic Programming (GP) can be used to solve a large number of complex problems in various application domains. However, one obvious shortcoming of GP is that it usually uses a substantial amount of processing time to arrive at a solution. In this paper, we present the parallel implementations that can reduce the processing time by using a coarse-grained model for parallelization and an asynchronous migration. The problem chosen to examine the parallel GP is a mobile robot navigation problem. The experimental results show that superlinear speedup of GP can be achieved.

1 Introduction

Genetic Programming was successfully used to perform automatic generation of mobile robot programs [1]. The use of the perturbation to improve robustness of the robot programs was proposed. Each robot program was evaluated under many environments that were different from the original one. As a result, the substantial processing time was required to evaluate the fitness of the robot programs.

To reduce the computational time, this study proposed two parallel implementations. Asynchronous and synchronous parallelization approaches were examined. We also compared the quality of the solutions generated from the serial and parallel GP.

The earlier work of parallel GP was implemented on a network of transputers by Koza and Andre [2]. Their result showed that the parallel speedup was greater than linear. Dracopoulos and Kent [3] proposed the use of the Bulk Synchronous Parallel Programming (BSP) model to parallelize genetic programming. Two approaches of parallel GP were examined on a cluster of Sun workstations. The first was based on a master-slave model while the second was based on a coarse-grained model. The results showed that the achieved speedup was close to linear. A recent paper by Punch [4] presented the empirical study about some problem-specific factors which affect the effectiveness of parallel GP. Punch concluded that the achieved performance of parallel GP by using a coarse-grained model may vary according to the nature of problems.

The remaining sections are organized as follows: The next section is a description of the mobile robot navigation problem. Section 3 describes a problem representation in the serial GP. Section 4 shows the parallel solutions. Section 5 presents the experimental results and discussion. Finally, section 6 provides the conclusions of this work.

2 Mobile Robot Navigation Problem

Our previous work [1], GP was used to generate a robot control program for the mobile robot navigation problem. The task was to control a mobile robot from a starting point to a target point in a simulated environment. The environment was filled with the obstacles which had several geometrical shapes.

The aim of the work was to generate *robust* control programs. In the evolution process, each individual was evaluated under many environments that were different from the original one. The result showed that the robustness of the robot programs was improved by such an approach.

3 Serial Algorithm

The terminal set is composed of three primitive movement controls {`move, left, right`} and one sensor information {`isnearer`}. The function set is composed of three functions {`if-and,if-or,if-not`}. The GP parameters are shown in Table 1.

The fitness function is a sum of the fitness value in each environment which is based on the distance of the final position and the number of moves.

In the evolution process, the percent of disturbance is 20% and the number of training environments is 8.

Table 1: GP parameters

Total population	6000
Crossover probability	0.9
Mutation probability	0.1
Reproduction	5% of Total population
Maximum generation	200

Table 2: Experimental Parameters

	Num. of Processors				
	1	2	4	6	10
Pop. size *	6000	3000	1500	1000	600
Environments *	8	7	4	3	2
Migration interval	NA	100	50	34	20

* per node

4 Parallel Genetic Programming

In a coarse-grained model, the population is divided into subpopulations and are maintained by different processors. The model is also known as *Island model* and the subpopulation is called *deme* [2].

Some works in parallelization of GA and GP using a coarse-grained model [2, 5] show that the results can achieve *superlinear speedup*[1]. This is caused by two factors; the speedup from the populations distributed across different processors and the speedup obtained by increasing the probability in finding the correct solution, as the number of populations is increased.

In applying the parallel approach to the previous work [1] by using a conventional coarse-grained model, the result achieves only linear speedup [6] since the amount of work is fixed – the algorithm is terminated when it reaches the maximum generation. Hence, the parallel algorithm does not exploit the probabilistic advantage that the answer may be obtained before the maximum generation. We reduce redundant jobs by dividing the *environments* among the processing nodes. After a specific number of generations, every subpopulations are migrated between processors using a fully connected topology. However, this scheme leads to the reduction of robustness since each individual has a shorter period in each training environment. To mend this problem, we increase the number of environments in each node. However, the number of environments per node should be less than the number of environments per node in the general coarse-grained model.

We implement our parallel algorithm on a dedicated cluster of PC workstations with 350 MHz Pentium II processors, each with 32 Mb of RAM, and running Linux as an operating system. These machines are connected via 10 Mbs ethernet cabling. We extend the program used in [1] to run under a clustered computer by using MPI as a message passing library.

Several trials are examined to find an appropriate value for the number of environments per node (see Table 2). The migration is carried out as follows: each node broadcasts its subpopulation to all other nodes by `MPI_Bcast` function, this is repeated for every node. The top 5% of individuals from each subpopulation are exchanged during the migration. Pseudo-code for the migration is shown in figure 1. The detail will be discussed in the timing analysis section.

```
procedure Migration
begin
    barrier[1] wait all nodes ready
    for i = 1 to n
    begin
        if (my_process_id = i)
            broadcast send
        else
        begin
            broadcast receive
        end
        barrier[2] wait for the next broadcast
    end
end
```

Figure 1: The migration process

The total population is hold constantly for the task and is divided equally among workstations. The number of selected individuals, crossover operation, mutation operation, reproduction are a percentage of the amount of the total population. The parallel efficiency is measured by varying a number of nodes and the results are averaged over 20 runs for each number of nodes.

In the first implementation, the migration between subpopulations is synchronized. Each node is blocked by `MPI_Barrier` function until all subpopulations evolve to the same number of generations. However, the synchronizing migration results in uneven work loads among the processors since the time required to complete the evaluation varies, with the least effective programs taking the longest period and the best programs taking the shortest period.

[1] *Superlinear speedup* means that speedup is greater than the number of processors used.

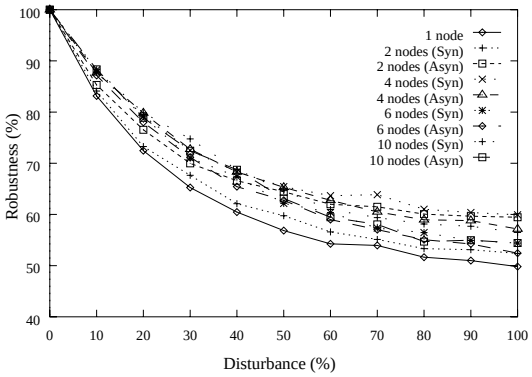

Figure 2: Robustness

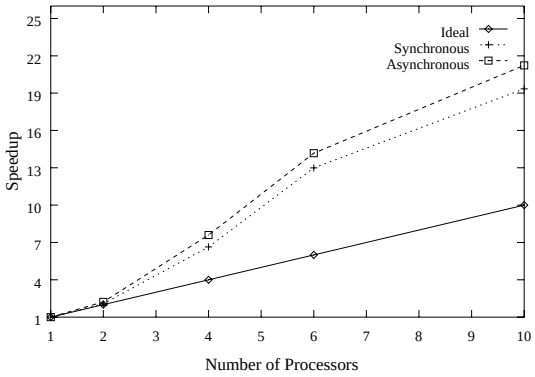

Figure 3: Speedup

In the second implementation, we attempt to further improve the speedup of the parallel algorithm by the asynchronous migration. When the fastest node reaches predetermined generation numbers, the migration request is sent to all subpopulations. The migration takes place at the end of the current generation. In this state, if any populations are still in the fitness evaluation phase, the other nodes must wait. The waiting time will be at most less than one generation.

5 Results and Discussion

5.1 Speedup

To make an adequate comparison between the serial algorithm and parallel algorithm, Cantú-Paz [7] suggests that the two must give the same quality of the solution. In this paper, we define the robustness of the generated program from the serial algorithm as a baseline. In addition, if the generated program from the parallel algorithm gives the same robustness as the program from the serial algorithm, the equal amount of work to achieve the same quality of the answer is done. From the robustness graph in figure 2, the generated program from the parallel GP is better than the serial GP. Hence, the amount of work from the parallel algorithm is not less than the serial algorithm.

The parallel speedup is defined as the ratio of the serial execution time to the parallel execution time.

$$Speedup = \frac{Serial\ time}{Parallel\ time} \quad (1)$$

Figure 3 illustrates the speedup observed on the two implementations as a function of the number of processors used. The performance is less than we expect, although both implementations exhibit superlinear speedup. The speedup curves taper off for 10 processors and the performance of the asynchronous implementation is slightly better than the performance of the synchronous implementation. In order to discern the cause of this result, the timing analysis is performed in the next section.

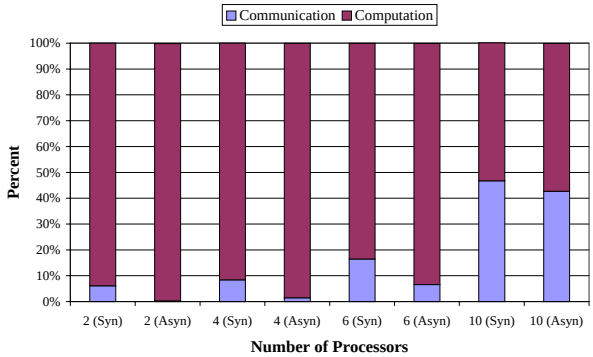

Figure 4: Percentage of time spent in computation and communication

5.2 Timing Analysis

Figure 4 shows the relative time spent in each section of the implementations. The communication overhead – the sum of the barrier time and broadcast time – goes up considerably as the number of processors increases. The asynchronous implementation does not help much on reducing the communication overhead at large numbers of processors. Thus, we investigate the further detailed analysis of the communication overhead.

Figure 5 shows the absolute time spent in major functions of the communication. The time spent in barriers indicates the time spent on waiting for all processes to reach the same point. From pseudo-code of the migration in figure 1, the barrier time consists of the time spent to wait for all nodes to be ready for the migration and the next broadcast.

In the synchronous implementation, the time spent in barriers reduces as the number of processors increases. This is because the barrier time depends on the variation

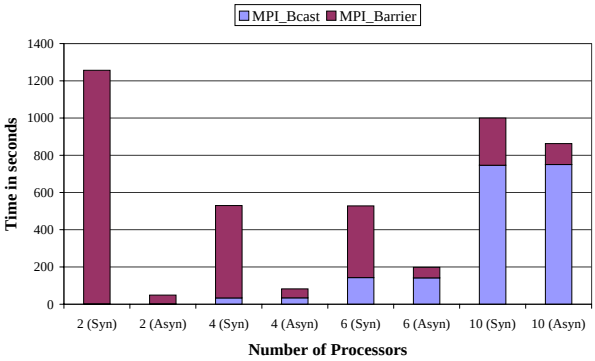

Figure 5: Absolute time spent in communication

of the computation time of each node. As the number of nodes is increased, the computation time per node is decreased. Hence, the barrier time is reduced.

In contrast, the barrier time in the asynchronous implementation increases as the number of processors increases. This is due to the fact that the time spent in the second barrier (waiting for the next broadcast) increases with the number of nodes. However, the asynchronous implementation eliminates the first barrier therefore it reduces the total time in the barriers compared to the synchronous implementation.

The absolute time spent in a broadcast increases considerably – greater than linear. From the inspection in the trace information by using a visualization tool, we found that the transmission of the broadcast functions in the implementation of MPI that we use may be executed more than once, especially for a large number of processors.

After obtaining some timing analyses, the results reveal the cause of the problem. The performance degradation in 10 processors is caused by the excessive communication time due to the broadcast function. Although the asynchronous migration reduces the barrier time effectively compared to the synchronous migration, the increase in the communication time in 10 processors obliterates this advantage. In case of the small number of processors (2,4,6), the gain from the asynchronous migration is considerable as the evolution proceeds at the speed of the fastest node.

As the size of the work increases (i.e., the number of training environments increases), the serial and parallel computation time will be increased when the time spent in the communication is constant. If the ratio of the computation/communication can be kept large (large work load), then one can expect that the parallel performance will be improved.

6 Conclusions

The result presented in this paper shows a success of speeding up the Genetic Programming process by means of parallel processing. The parallel implementations of Genetic Programming successfully exploit the computing resource of a dedicated cluster of PC workstations. Superlinear speedup of GP can be acquired by improving a coarse-grained model for parallelization as less computational work needs to be done. Furthermore, the timing analyses indicate the scalability of the parallel approaches, as the size of the problem increases, the speedup will be improved.

References

[1] Chongstitvatana P (1998), Improving robustness of robot programs generated by genetic programming for dynamic environments. Proc. of IEEE Asia Pacific Conference on Circuits and Systems, p.523–526

[2] Koza JR, Andre D (1995), Parallel genetic programming on a network of transputers. Proc. of the Workshop on Genetic Programming: From Theory to Real-World Applications, University of Rochester, National Resource Laboratory for the Study of Brain and Behavior, Technical Report 95-2, p.111-120

[3] Dracopoulos DC, Kent S (1996), Bulk synchronous parallelisation of genetic programming. Proc. of the Third International Workshop on Applied Parallel Computing in Industrial Problems and Optimization (PARA '96), Springer Verlag, Berlin

[4] Punch B (1998), How effective are multiple poplulations in genetic programming. Proc. of the Third Annual Conference in Genetic Programming, pp.308-313

[5] Lin S-C, Punch WF, Goodman ED (1994), Coarse-grain parallel genetic algorithms: Categorization and new approach. Proc. of the Sixth IEEE SPDP, pp.28-37

[6] Tongchim S, Chongstitvatana P (1999), Speedup Improvement on Automatic Robot Programming by Parallel Genetic Programming. Proc. of 1999 IEEE International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS'99), Phuket, Thailand

[7] Cantú-Paz E (1999), Designing efficient and accurate parallel genetic algorithms. PhD thesis, University of Illinois at Urbana-Champaign