Shared Memory Genetic Algorithms in a Multi-Agent Context

Dana Vrajitoru
Intelligent Systems Laboratory
Computer and Information Sciences
Indiana University South Bend
South Bend, IN 46634, USA
danav@cs.iusb.edu

ABSTRACT
In this paper we present a concurrent implementation of genetic algorithms designed for shared memory architectures intended to take advantage of multi-core processor platforms. Our algorithm divides the problems into sub-problems as opposed to the usual approach of dividing the population into niches. We show tests for timing and performance on a variety of platforms.
Track: Parallel Evolutionary Systems

Categories and Subject Descriptors
I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence, Multiagent Systems

General Terms
Algorithms

Keywords
Parallel genetic algorithms, shared memory, multi-agents

1. INTRODUCTION
The hardware developments from recent years have shown a clear tendency towards improving the CPUs performance by increasing the number of cores. Thus, dual core architectures have become common place in the industry and even quad core computers are now available to the large public. The issue we are now facing is taking advantage of such platforms and writing programs that make use efficiently of the multiple CPU cores.

The shared-memory genetic and evolutionary computing algorithms are closely related to the parallel and distributed models. The parallelization techniques can be ported from one type of architecture to the other, with the only difference being the APIs being used. Each architecture and library can offer unique opportunities for the optimization of the execution time. A survey of these algorithms can be found in [3], [5]. In a multi-core context, a shared memory API is expected to provide a faster and more direct way for the processes to communicate and synchronize and avoid the overhead of a message-passing API.

Parallel and distributed versions of the genetic and evolutionary algorithms are popular and diverse. The simplest parallel models are function-based where the evaluation of the fitness function is distributed among the processes. The most popular parallel models are population-based where the population itself is distributed in niches [1], [7], [9], [11]. Such models require a periodic migration of individuals between the sub-populations, and the influence of migration patterns has also been an object of study [4], [15].

Our model is based on a division at the genotype level of the population into several agents or processes. Each process then receives a partial chromosome to evolve. All the genetic operations are then restricted to this subset of genes. For evaluation purposes, a template is kept by every process containing information about the best genes found by all of the other processes up to that point. A periodic exchange procedure keeps this information up to date.

Although a division of the population among the processes is a more popular approach and certainly defendable, we believe that making the split along the genotype can also present some advantages. In a previous study [12], we have investigated the role of the population size versus the number of generations and concluded that a larger population can lead to better results. Splitting the chromosome among the processes allows each of them to work with a larger population while having to focus on a smaller number of genes. The latter should allow a faster evolution for those genes if the fitness is defined well enough.

Similar ideas to our parallel model of genetic algorithms have been used in [10] to distribute the compact genetic algorithm in a message-passing architecture. The model proposed in [8] also distributed the chromosome another the processes, and combines this with a temporal distribution of the solutions.

Multiple cores have started to be used in genetic and evolutionary computations as well. For example, [2] distributes the evolutionary operations themselves among threads executed on four cores, while the population remains whole and is shared by the threads without distribution.

Section 2 presents our multi-threaded model of genetic algorithms designed for shared memory platforms. Section 3 introduces two two test problems we have used for our model. Section 4 presents our experimental results, and we finish the paper with some conclusions.

2. MULTI-AGENT PARALLEL MODEL

Our model for parallel genetic algorithms follows a similar idea to the one described in [13]. The difference is that the current model is implemented for a shared memory architecture as opposed to a Beowulf cluster, and the experiments use a different set of problems.

2.1 Problem Division

According to the most popular approach to parallel genetic algorithms, which is the nested one, the population is decomposed in several nests or niches, each of them evolving in parallel. In such a model, the evolution in each population is self-contained, and the only thing that makes it a unified process is an occasional migration of individuals between the nests.

In the approach proposed in this paper, all of the chromosomes are divided among the processes, such that the task of each process consists in evolving part of the chromosome. Figure 1 illustrates this concept. In some cases, the fitness function is of such nature that the problem to be solved can also be divided into several sub-units conceptually, while in other cases this division is purely artificial. We do not make a distinction between the two situations in our model.

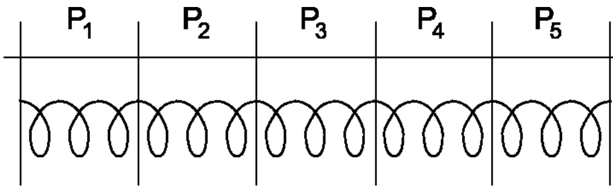

Figure 1: The division of the chromosome among the processes or agents

All the genetic operators are applied just as usual, except that they are restricted to a subset of the genes that was assigned to each process. Since the evaluation of the fitness function usually requires the entire set of genes, the process will have access to a template set for the part of the chromosome that is not under its control. This template is periodically exchanged between the processes. Figure 2 illustrates this idea.

The idea behind this parallel model is that the problem to be solved is divided into several tasks. Each task is then assigned to a different agent that will focus on it while exchanging information with the other agents. The multi-agent approach is one that has proved efficient for many applications before, in a variety of contexts.

Let n be the size of the chromosome, with the indexes for the genes going from 0 to $n-1$. Let us suppose that we have p processes. Each agent/process will receive a part of the chromosome of size $n_p = n/p$. Then the process or agent

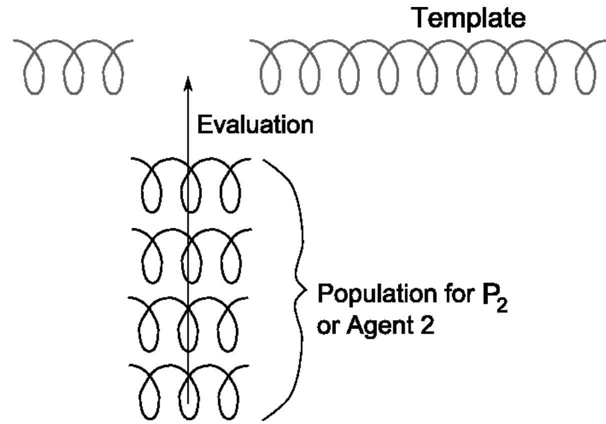

Figure 2: A template for the population evolved by one of the agents

number id will be in charge of the genes between the indexes $(id-1) * n_p$ and $id * n_p - 1$.

2.2 Fitness Evaluation

In the literature about genetic algorithms one can find a good number of examples where the fitness function can be nicely divided into several sub-problems such that the evaluation of each of them can be accomplished independently. Our model does not focus on these types of problems specifically. It is designed in a general way such that it can be applied to any fitness function. However, the evaluation process can be sped up for these special cases and a greater performance can be achieved in terms of execution time and use of each CPU core. This might be the subject of future research.

We start from the assumption that to evaluate the fitness function for any combination of genes, we need a full set spanning from 0 to $n-1$. Thus, to evaluate a partial chromosome, we need to complete it with a sample of the genes that it does not contain. We call this sample a *template*, and each process will hold one in memory. The evaluation consists in plugging the partial chromosome into the common template, and then passing this complete individual to the fitness function.

An exchange procedure insures that the template is kept reasonably up to date with respect to the latest best performing genes obtained by each agent. During the exchange phase, each process copies the genes of the best chromosome found so far in terms of fitness to a common "best chromosome" which is shared by all the processes. After all of the processes have finished this update, each of them makes a copy of the genes in the best chromosome belonging to all the other agents. This becomes then the new template for each process.

The exchange procedure is shown in Figure 3 in C++ based pseudocode. In this algorithm we assume that the indexes in the partial chromosome are kept consistent with the position of the genes in the complete chromosome. Just to make the procedure easier to understand, the *id* of the process is used as an index for the best partial chromosome and

for the template. Practically, our implementation is object oriented, the exchange function is a class method, and these objects are class attributes.

```
void Exchange(int id) { // the process
  np = chromosome_size/ number_of_proc;
  Barrier(number_of_proc);
  for (i=(id-1)*np; i<id*np; i++)
    best_chromosome[i] = best_partial_chr[id][i];
  Barrier(number_of_proc);
  for (i=0; i<(id-1)*np; i++)
    template[id][i] = best_chromosome[i];
  for (i=id*np; i<n; i++)
    template[id][i] = best_chromosome[i];
}
```

Figure 3: The best chromosome exchange procedure

Technically the exchange process is the only synchronization required between the processes. For this we employ a barrier after the first phase of the exchange, to make sure that all the processes have finished updating the best chromosome before they start copying the information from it into their own templates.

The population is initialized for each process randomly, as is usually the case. The template is initialized by calling the function exchange before the evolution process starts.

The exchange takes place every few generations, 10 for most of our experiments. One of the questions we shall attempt to answer is how much it interferes both with the execution time and with the performance in terms of best fitness achieved.

Given the fact that the number of variables was divisible by the number of processes that we've employed in all the cases, we were able to implement a small optimization to our algorithm. Thus, for each process, the variables belonging to the template do not need to be converted from their binary representation to real values every time, and we can simply store and reuse their real values in between the exchange phases. Experimentally this has shown a small improvement for one of the functions where the fitness is not computationally expensive, but has no visible impact for the second one.

3. PROBLEMS

We have chosen two completely different problems to test our parallel model and the specifics of each of them should allow us to showcase different features of our program. These two problems have a common feature, which is the fact that they use a given number of variables taking real values. For both problems we have used 36 variables and 10 binary genes to represent each of them.

3.1 Polynomial Problem

The first problem is a simple maximizing function where the evaluation of the function itself is not very costly. Thus, we need to find values for the variables x_i that maximize the following polynomial:

$$F_p(x_1, x_2, \dots x_m) = \sum_{i=0}^{m} (i+1) \cdot (16 - x_i^4),$$

where $-0.5 \leq x_i \leq 1.27,\ \forall i,\ 1 \leq i \leq m$.

This is a function that is not computationally expensive, and the experiments with timing the program using this function will tell us in particular how good the parallelization of the genetic operations themselves is. A second feature of this function is that its evaluation is uniform across the chromosomes from the point of view of the complexity, making it easier to parallelize.

The second problem that we are using is optimizing the parameters defining a pilot for a simulated motorcycle. Since this problem is more complex, we shall describe it more in detail in the next section. For this second problem, the evaluation requires significantly more computations, and thus it will allow us to observe the improvement in performance in that respect. Contrary to the first function, the complexity of evaluating a chromosome is not uniform, but can vary significantly from one individual to the next, This constitutes an additional challenge for the parallel model.

3.2 Autonomous Pilot for a Motorcycle

In this subsection we introduce the details of the simulated vehicle and its autonomous pilot, as well as the application of genetic algorithms to its configuration.

The physical model of the motorcycle has been more extensively described in [14] and is close to [6]. The motorcycle or STV, is modeled as a system composed of several elements with various degrees of freedom that can be driven through several control units. Figure 4 shows the components of the physical model for a motorcycle.

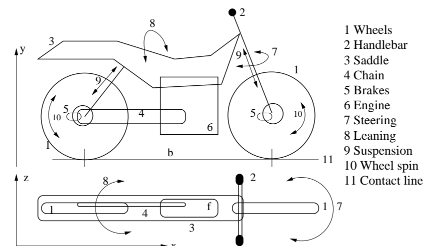

Figure 4: A motorcycle with control units and degrees of freedom

An STV is a non-holonomic dynamic system with six degrees of freedom: the rotation of the wheels around an axis parallel to Oz, the rotation of the handlebar and of the front wheel around the fork axis (steering), the front and back translation along the suspension axis, and the rotation of the whole vehicle around the Ox axis in a system of coordinates relative to the motorcycle where the origin is in the center of the vehicle, on the ground level. The driver can control the vehicle through five inputs: the handlebar steering, leaning the vehicle laterally, the throttle, and the two brakes, front and back.

The STV is modeled as a reduced state system of continuous

variables. The generalized coordinates of the vehicle at a particular moment are given by

$$q = (s, \alpha, \theta)^T \quad (1)$$

where $s(t) = (x(t), z(t))$ represents the *spatial position* of the STV, α the *leaning angle*, and θ the *orientation angle* determining the *direction of movement* $d = (\cos\theta, \sin\theta)$.

The vertical component of both s and d is determined by the altitude and by the slope of the road considering the current position and orientation of the vehicle. The altitude is considered low enough that the gravitational acceleration is the constant $g = 9.8\ m/s^2$. Let $\sigma(s, d)$ be the angle made by the contact line of the vehicle with the horizontal plane (x, z).

The driver's input into the system is defined by the tuple $u = (\tau, \beta_f, \beta_r, \phi, \alpha)$ where τ is the component of the acceleration tangent to the direction of movement d and β_f, β_r represent the front and rear brakes respectively. This driver can be either a human player or an autonomous agent controlling the vehicle.

Let $v = s'$ be the momentary speed or velocity in the direction of movement, and $a = v' = s''$ the momentary acceleration in the direction of movement. The motion of the vehicle is modeled using Newtonian mechanics. The position and velocity of the vehicle at $t + \Delta t$ are defined by

$$s(t + \Delta t) = s(t) + \Delta s, v(t + \Delta t) = v(t) + \Delta v \quad (2)$$

where:

$$\Delta s = d\left(v \cdot \Delta t + a\frac{\Delta t^2}{2}\right), \quad \Delta v = a \cdot \Delta t \quad (3)$$

The acceleration is defined by the gravity, the friction, the drag, and the throttle. The brakes do not act as a simple negative acceleration, but contribute to the friction force instead.

3.3 The Autonomous Pilot

In this subsection we present the multi-agent autonomous pilot for our motorcycle and the perceptual information it uses.

The autonomous pilot uses perceptual information to make decisions about the vehicle driving. This information is inspired from the perceptual cues that a human driver would also be paying attention to while driving a vehicle. In this application, the pilot is aware of the following measures:

The *visible front distance*, denoted by $front$, defined as the distance to the border of the road from the current position in the direction of movement, scaled by the length of the vehicle, or *horizon*.

The *front probes*, denoted by $frontl$ and $frontr$, are defined as the distances to the border of the road from the current position of the vehicle in directions rotated left and right by a small angle from the direction of movement.

The *lateral distances*, denoted by $leftd$ and $rightd$, are measures of the lateral distance from the vehicle to the border of the road, at a short distance ahead of the vehicle, simulating what the pilot might be aware of without turning their head to look.

The *slope* is a perceptual version of σ, discretized to simulate the intuitive notion of road inclination that a human driver would have, approximated by the values almost flat, slightly inclined up or down, or highly inclined up or down. This simulates the fact that a human pilot is not aware of the precise value of σ.

Figure 5 shows an example of the geometrical definition of these measures.

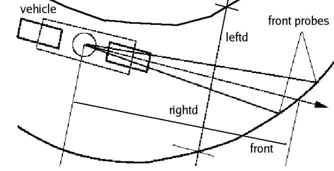

Figure 5: Perceptual information used by the autonomous pilot

The motorcycle is driven by several control units (CUs). Each of them is controlled by an independent agent with a probabilistic behavior. The agents are not active during the computation of each new frame simulating the evolution of the vehicle on the road, but only once in a while in a non-deterministic manner. This simulates the behavior or a human driver that may not be able to respond instantly to the road situation and requires some reaction time.

The current control units focus on the gas (throttle), the brakes, the handlebar/leaning. Each of these CUs is independently adjusted by an agent whose behavior is intended to drive the motorcycle safely in the middle of the road at a speed close to a given limit. In our case, the agents controlling the throttle and the handlebar are in general more active than the agent controlling the brakes.

The agents behave based on a set of equations relating the road conditions to action. The full set of equations is described in [14]. Here we will briefly describe each of the agents. The equations comprise a fair number of coefficients and thresholds. The configuration of each agent uses independent values for the coefficients.

The Throttle. This agent controls the amount of gas supplied to the engine and thus the speed of the vehicle.

The agent uses a minimal speed threshold v_{low}, a maximal speed threshold over which the speed is considered unsafe, and the given speed limit v_{limit}. The agent aims to keep the vehicle speed above v_{low} and below the maximal one, and also close below the v_{limit}.

The agent detects a turn in the road by testing $leftd$ and $rightd$ and if needed, cuts the gas to allow for a safe turn. A similar rule is applied to the visible distance in front of the driver: a low value for $front$ indicates an unsafe road situation requiring a reduced speed. In any other situation it attempts to keep the speed close to v_{limit}.

The Brakes. The agent controlling the brakes presents a similar behavior to the throttle agent but can only decrease the speed. This agent acts when a more dramatic change is necessary.

The Steering / Leaning Agent. The motorcycle can achieve a change in direction either by steering using the handlebar, or by leaning. The autonomous pilot has been tested under three conditions: when the motorcycle is driven entirely by steering, when the motorcycle is driven entirely by leaning, and a combined strategy, consisting of steering at low speed (bellow a threshold) and leaning at a speed above the threshold. This emulates the general strategy employed by human drivers.

Alerting Agent. Beside all the agents that are in direct control of the motorcycle, the pilot comprises a fourth agent that does not perform any action on the vehicle. While the other agents are active only occasionally, this agent is probing the vehicle and road conditions for every new frame and is capable of activating one of the other agents if the situation requires special attention. Such situations include the speed of the vehicle being too high or too low, or the visible front distance being too short.

3.4 Pilot Configuration by Genetic Algorithms

All of the agents composing the autonomous pilot are governed by equations comprising a set of thresholds and constants that can be configured to adjust its behavior and optimize its performance.

To apply the GAs to this problem, we chose a representation where each configurable coefficient is assigned 10 binary genes, and the chromosome results by concatenating all of the coefficients. Thus, we worked with 36 coefficients because the pilot combines the leaning and steering modes. This means that the chromosome is of a length of 320 and 360 respectively.

We used the one-point crossover for our experiments with a probability of 0.8 and a probability of mutation of 0.01. We employed an elitist reproduction preserving the best individual from each generation to the next.

A chromosome is evaluated by running the motorcycle in a non-graphical environment once with the pilot configured based on values obtained by decoding the chromosome over a test circuit presenting various turning and slope challenges. To compute the *objective fitness* we marked 50 reference points on the road and counted how many of them were almost touched by the motorcycle. The fitness is computed as follows:

$$F(x) = \frac{d_m}{d_t} + \frac{1}{1 + t_m} \quad (4)$$

where d_m is the number of points crossed by the motorcycle, d_t is the total number of points, and t_m is the total time taken until either the circuit was completed, or until a failure condition was detected.

Thus the objective fitness reflects both how much of the circuit the motorcycle completed, and how fast it was capable of finishing the track. In general, a fitness higher than 1 is an indication of completion of the circuit.

Table 1: Technical specifications of the platforms used for testing

Label	CPU Make	CPU Speed	Core	OS
U1	Intel Pentium 4	2.8 GHz	1	Ub
M2	Intel Core 2 Duo	2.4 Ghz	2	OsX
X2	Intel Dual Core T7200	2.0 GHz	2	XP
X4	Intel Quad Core Q6600	2.4 GHz	4	XP

Table 2: Timing in seconds of the polynomial function in 1000 generations, average over 100 runs

Platform	#Processes			
	1	2	4	8
U1	2.41	3.77	5.56	13.17
M2	0.7	1.52	1.34	2.07
X2	1.82	1.24	2.71	4.87
X4	1.26	0.77	0.53	1.42

A failed circuit can be caused by one of the following three situations: a crash due to a high leaning angle, an exit from the road with no immediate recovery, or crossing the starting line without having reached all the marks, as when the vehicle takes a turn of 180 degrees and continues backward.

4. EXPERIMENTAL RESULTS

In this section we present some of the experimental results testing both the executions time/speedup and the fitness performance.

We have performed a set of experiments for each of our two problems that consisted on a number of runs: 100 for the polynomial problem, 10 for the motorcycle one, on four different platforms. The starting population was the same for each platform, but different for each run. We've let the evolution run for 1000 generations in each case and we have measured the execution time and the fitness achieved in all the cases. Since the platforms present different operating systems and numbers of CPU cores, testing the program in these various conditions will tell us how efficient the implementation is on each of them, as well as how efficient each operating system is. In terms of fitness it shouldn't matter which platform the program is tested on.

Table 1 introduces the four platforms that we have used for our computations. In the operating system column, XP stands for Windows XP, OsX stands for the Mac OsX 10.5.6, and Ub stands for Ubuntu 8.04.

Table 2 shows the average execution time in number of seconds of the polynomial function. The chromosome length is 360, the population is of size 50, and we have run 1000 generations in all the cases. The results are averaged over 100 runs.

To complement these timing results, Table 3 shows the speedup in all of these cases computed as the execution time on a

Table 3: Speedup for the polynomial function computed as the sequential time divided by the parallel time based on Table 2

Platform	#Processes		
	2	4	8
U1	63.93%	43.35%	18.30%
M2	46.05%	52.24%	33.82%
X2	146.77%	67.16%	37.37%
X4	163.64%	237.74%	88.73%

Table 4: Timing in seconds of the motorcycle driving, 1000 generations, average over 10 runs

Platform	#Processes			
	1	2	4	8
U1	1969.3	6488.0	13308.1	32031.9
M2	1789.7	7459.4	8396.22	12882.0
X2	1292.5	2922.7	9542.0	10301.3
X4	2081.0	3054.7	5452.5	16955.7

single process divided by the execution time of each multi-threaded run. A speedup of more than 100% represents a faster execution time in parallel than sequentially. Note that the speedup for 8 processes is not expected to be improved on any of the platforms, since the maximum number of cores that were available on any of the machines is 4. This table shows that on most architectures where several cores were available, the execution time for a number of processes less or equal to the number of core presents a speedup. These results showcase the type of problem where the genetic operations themselves might take a significant amount of computational time compared to the evaluation of the fitness function.

Table 4 shows the timing in seconds for the motorcycle driving problem. The settings in terms of chromosome length, population size, and number of generations are exactly the same as for the polynomial function. Since the evaluation itself can take a variable amount of time depending on how long the pilot survives on the road, we have also recorded the total number of times that the function move has been called for the motorcycle during the evaluation. We can consider these calls to be basic operations because they require a uniform amount of time. Since the function move is called repeatedly until either a crash condition occurs, or until the vehicle finishes the track, it is the number of such calls that introduces a great variety in the evaluation time. Thus, this measure tells us the number of operations executed in every case. Table 5 shows the number of such calls divided by 10^4 as an average over 10 runs.

The synchronization protocol requires the threads or agents to periodically wait for each other. Since the complexity of the fitness evaluation is not uniform, this means that every 10 generations each of them will be as slow as the slowest of them. This observation explains why the speedup for the motorcycle problem is not as straightforward on the Quad-Core architecture (X4) as it is the case for the polynomial function. As future research we intend to find out how necessary this synchronization is. Thus, we could allow the threads to evolve at their own pace and examine how the general performance is affected.

Table 5: Total number of moves for the motorcycle driving, divided by 10^4, average over 10 runs

Platform	#Processes			
	1	2	4	8
U1	18678.0	4180.98	5849.64	11063.75
M2	1881.98	6578.59	7344.52	13074.47
X2	2063.64	5856.67	18086.01	19882.85
X4	4051.34	7763.57	5942.87	14961.18

Table 6: Computational time in seconds for 10^4 moves

Platform	#Processes			
	1	2	4	8
U1	1.05	1.55	2.28	2.90
X2	0.63	0.50	0.53	0.52
X4	0.51	0.39	0.92	1.13
M2	0.95	1.13	1.14	0.99

To compute the speedup for this problem, we need to compute a baseline of comparison that does not depend on the number of operations executed for each case. Thus, Table 6 computes the average time is seconds for every 10^4 move operations by dividing the time in Table 4 by the number of moves in Table 5. Then Table 7 shows the speedup for this problem by dividing this new timing measure for the sequential case by its value for the parallel case.

Finally we need to observe the average fitness achieved after 1000 generation for both problems to see if the parallel model can perform just as well as the sequential one. Table 8 shows these results for the polynomial function first, then for the motorcycle pilot. We have denoted the fitness function for the polynomial problem $F_p - 10600$ because all the

Table 7: Speedup for the motorcycle pilot problem computed as the sequential time divided by the parallel time in Table 6

Platform	#Processes		
	2	4	8
U1	67.94%	46.34%	36.41%
M2	83.87%	83.19%	96.52%
X2	125.51%	118.71%	120.89%
X4	130.55%	55.99%	45.32%

Table 8: **Average fitness after 1000 generations**

Problem	#Processes			
	1	2	4	8
F_p-10600	54.92	55.86	55.99	56.0
moto	0.7999	0.9999	0.9165	0.7950

Table 9: Average execution time (s) for 1000 generations as the process communication/synchronization period varies

	#Processes			
Synch. Period	1	2	4	8
1	1.28	0.89	0.71	1.96
10	1.26	0.77	0.53	1.42
50	1.27	0.76	0.51	1.07
100	1.26	0.76	0.51	1.01
500	1.26	0.76	0.52	0.86
1000	1.27	0.76	0.51	0.80

values achieved for this problem were higher than this number, so we decided to subtract it from the reported fitness for compactness of the table and to make it easier to read. Larger numbers are better in both cases. We can see that the parallel model outperforms the sequential one in terms of fitness for both problems. For the polynomial problem it seems that even a higher division of the genotype among the processes is not detrimental to the performance. On the other hand, for the motorcycle pilot, a higher division of the genotype doesn't help improve the performance further.

This fact can be explained by the fact that the polynomial function, is more separable, while for the motorcycle pilot, one set of genes can influence the performance of a separate set. For example, a good function for the change in direction cannot easily be found if the function adapting the speed to the road conditions is not adequate.

Table 9 show an experiment where we varied the number of generations after which the different threads exchange their best individuals periodically. We have used the polynomial function and the X4 platform (QuadCore) for this purpose. The exchange period appears in the first column. This affects the amount of necessary synchronization between the processes and thus the execution time. The total number of generations is 1000, so a period of 1000 represents almost no synchronization, while a period of 1 represents a complete synchronization done after each new generation is built. The time in seconds represents the average execution time over 100 trials with a population of size 50. For a single process, the time should be the same in all the cases, since the exchange is not applicable. The small differences can be attributed to background OS processes interfering with the computations.

This table suggests that the communication does not interfere much with the execution time beyond a period of 10

Table 10: Average fitness for the polynomial function -10600 for 1000 generations as the process communication/synchronization period varies

	#Processes			
Synch. Period	1	2	4	8
1	54.92	55.85	55.99	55.09
10		55.86	55.99	56.0
50		55.83	56.00	56.00
100		55.81	56.00	56.00
500		55.36	55.97	55.99
1000		18.45	-12.86	-21.46

generations. To complement these results, Table 10 summarizes the average fitness obtained after 1000 generation with each synchronization period. The table shows that a communication step of 10 yields a similar performance to a communication step of 1, or even better. The performance remains good up to a step of 100 generations, but decreases significantly when it is up to 1000. This means that the communication period of 10 generations that we have chosen will not affect the execution time significantly, while also providing a good performance in terms of fitness.

5. CONCLUSIONS

In this paper we presented a multi-agent parallel model of genetic algorithms designed to take advantage of multiple CPU cores in a shared memory architecture. We have tested our model with two sets of problems of various difficulty and on four different platforms with several types of processors and operating systems.

In Section 2 we introduced our parallel model that divides the genotype among parallel threads or agents on a shared memory architecture. In this section we also introduced a polynomial optimization problem that we used for the test. Section 3 presented the test problems, a polynomial one and configuring an autonomous pilot for a simulated motorcycle. This constitutes a hard problem to solve as the evaluation function is complex and costly.

The experimental results presented in Section 4 explore the performance of the parallel model on several levels. First, in what concerns the *speedup*, for the polynomial function there is a clear improvement on the platforms with multiple CPUs. A more modest but still noticeable speedup can be observed as well for the more difficult problem of configuring the autonomous pilot. This is due to the need for synchronization and to the non-uniformity of the fitness evaluation in terms of execution time.

In terms of average *fitness* achieved in 1000 generations, for both test problems we can observe that the parallel model outperforms the sequential model for a number of processes less or equal to 4, which is also the maximum number of available cores on our test platforms.

Finally, a set of experiments has shown that the synchronization and communication period of 10 generations that

we have chosen has little impact on the execution time, but allows the fitness to achieve better levels than the sequential model.

In conclusion, our model presents a valid approach to taking advantage of the computing technologies that are becoming widely available.

6. REFERENCES

[1] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002.

[2] B. Baker, C. Carter, and G. Dozier. Sema: A new paradigm for distributed genetic and evolutionary computating. In B. Kim, editor, *Proceeding of the Midwest Artificial Intelligence and Cognitive Science Conference*, pages 35–39, Fort Wayne, IN, 2009.

[3] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998. Paris: Hermes.

[4] E. Cantú-Paz. Migration polices, selection pressure, and parallel evolutionary algorithms. *Journal of heuristics*, 7(4):311–334, 2001.

[5] G. Dozier. A comparison of static and adaptive replacement strategies for distributed steady-state evolutionary path planning in non-stationary environments. *International Journal of Knowledge-Based Intellident Engineering Systems (KES)*, 7(1):1–8, January 2003.

[6] N. Getz. Control of balance for a nonlinear nonholonomic no-minimum phase model of a bicycle. In *American Control Conference*, Baltimore, June 1994.

[7] K.B. Harvey and C.C. Pettey. The outlaw method for solving multimodal functions with split ring parallel genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 274–280, Orlando (FL), 1999. Morgan Kaufmann Publishers.

[8] M. Kim, V. Aggarwal, U. O'Reilly, and M. Médard. A doubly distributed genetic algorithm for network coding. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1272–1279, London, UK, July 7-11 2007. ACM.

[9] X. Llorà, R. Reddy, B. Matesic, and R. Bhargava. Towards better than human capability in diagnosing prostate cancer using infrared spectroscopic imaging. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 2098–2105, London, UK, July 7-11 2007. ACM.

[10] K. Sastry, D. Goldberg, and X. Llorà. Towards billion-bit optimization via a parallel estimation of distribution algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 577–584, London, UK, July 7-11 2007. ACM.

[11] I. Sekaj and M. Oravec. Selected population characteristics of fine-grained parallel genetic algorithms with re-initialization. In *GEC '09: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 945–948, New York, NY, USA, 2009. ACM.

[12] D. Vrajitoru. *Soft Computing in Information Retrieval. Techniques and Applications*, chapter Large Population or Many Generations for Genetic Algorithms? Implications in Information Retrieval, pages 199–222. Physica-Verlag, Heidelberg, Germany, 2000.

[13] D. Vrajitoru. Parallel genetic algorithms based on coevolution. In R.K. Belew and H. Juillé, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, Late breaking papers*, pages 450–457, 2001.

[14] D. Vrajitoru and R. Mehler. Multi-agent autonomous pilot for single-track vehicles. In *Proceedings of the IASTED Conference on Modeling and Simulation*, Oranjestad, Aruba, 2005.

[15] K. De Jong Z. Skolicki. The influence of migration sizes and intervals on island models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1295–1302, Washington DC, USA, June 25-29 2005.