Asynchronous Multi-Threaded Model for Genetic Algorithms

Dana Vrajitoru
Indiana University South Bend
Computer and Information Sciences
South Bend, IN 46617

Abstract

In this paper we present a parallel implementation of genetic algorithms using a shared memory model designed to take advantage of multi-core processor platforms. Our algorithm divides the problems into subproblems as opposed to the usual approach of dividing the population into niches. We propose an approach where the threads do not have to synchronize their evolution at any level and compare it with a synchronized model. We experiment with the timing and with the achieved fitness on a several platforms.

1. Introduction

As CPU speed seems to have reached a peak in its development, the hardware industry is currently shifting towards multiple CPUs to continue to improve performance. Thus, dual core architectures have become commonplace 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 these multiple CPU cores.

The shared-memory genetic and evolutionary computing algorithms follow similar implementation patterns to the parallel and distributed models. The parallelization techniques can be ported from one type of architecture to the other, with the 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 (Cantú-Paz 1998), (Dozier 2003).

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 (Alba and Tomassini 2002), (Harvey and Pettey 1999), (Llorà et al. 2007), (Sekaj and Oravec 2009). 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 (Cantú-Paz 2001), (Z. Skolicki 2005).

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.

Similar ideas to our parallel model of genetic algorithms have been used in (Sastry, Goldberg, and Llorà 2007) to distribute the compact genetic algorithm in a message-passing architecture. The model proposed in (Kim et al. 2007) also distributed the chromosome another the processes, and combined this with a temporal distribution of the solutions.

Multiple cores have started to be used in genetic and evolutionary computations as well. For example, (Baker, Carter, and Dozier 2009) distributes the evolutionary operations themselves among threads executed on four cores, while the population remains whole and is shared by the threads without distribution.

2. Multi-Threaded Model

Our model for parallel genetic algorithms follows a similar idea to the one described in (Vrajitoru 2001). 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 fitness landscape.

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.

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

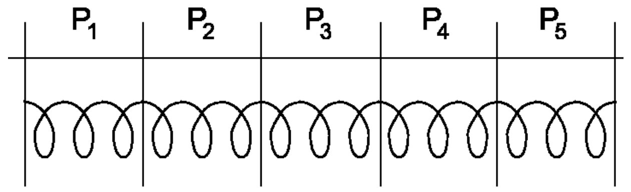

Figure 1: The division of the chromosome among the processes or agents

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.

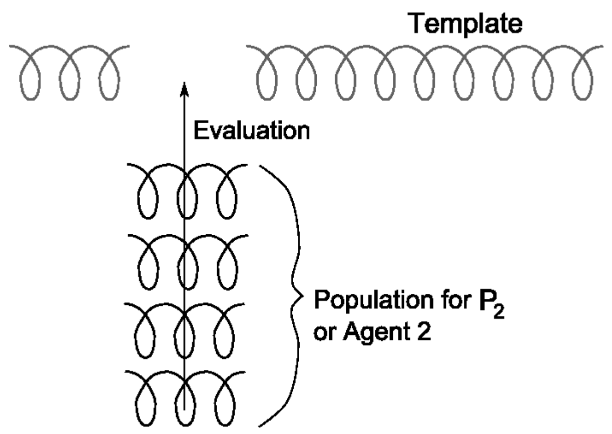

Figure 2: A template for the population evolved by one of the agents

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 been 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 process will receive a part of the chromosome of size $n_p = n/p$. Then the process 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 on genetic algorithms there are 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 procedure 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 its memory. The evaluation consists of 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. 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 local copy of the genes in the best chromosome belonging to all the other agents. This becomes then the new template for each process. Keeping the template as a local copy for each process even though the information is redundant allows the process to use it without a need to protect the read and write operations.

2.3 Synchronous vs Asynchronous Exchange

The main part of the communication between the processes happens during these exchange procedure. This is also the fundamental difference between the two models that we compare here. Let us suppose that the communication period is 10 generations. In both cases the exchange will consist of a writing phase and a reading phase.

In the *synchronous model*, every 10 generations every process proceeds to the writing phase, then gets into a Barrier. Thus, before they can continue to the read phase, all of the processes must finish the exact same number of generations and supply the best chromosome they have found at that point. Thus, the evolution of each subpopulation proceeds relatively in synch with all of the others. Since the execution time required by a call to the fitness evaluation function can vary quite a great deal in our case, this means that every 10 generations all the processes must wait for the slowest of them to catch up. This can impact the overall speedup from parallelizing the algorithm.

In the *asynchronous model*, after each 10 generations, each process first writes its best chromosome into the common memory, and then proceeds to retrieve the information supplied by all the other processes from the same common place without having to wait. Thus, the best common chromosome might contain at any moment the best genes found by each process after a completely different number of generations. Since no waiting is required, we expect the computational time to improve, as well as the speedup on multi-core architectures. The main question that we try to answer in this paper is whether this discrepancy between the processes will impact the achieved fitness in a negative way.

The exchange procedure is shown in Figure 4 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 *proc_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 Synch_Exchange(int proc_id) {
  np = chromosome_size/ number_of_proc;
  for (i=(id-1)*np; i<id*np; i++)
    best_chrom[i] =
               best_partial_chr[id][i];
  Barrier(number_of_proc);
  for (i=0; i<(id-1)*np; i++)
    template[id][i] = best_chrom[i];
  for (i=id*np; i<n; i++)
    template[id][i] = best_chrom[i];
}
```

Figure 3: Synchronous exchange procedure for the best chromosome

In the algorithm in Figure 4, the memory locations used for the exchange do not need to be protected. As it happens, before the barrier each process is writing information in its own part of the best chromosome and they do not interfere with each other. After the barrier several threads need to read from the same location, but we have adopted the assumption that multiple simultaneous read operations can be allowed. Practical tests have shown that these operations are performed correctly indeed. The Barrier separates the write phase from the read phase, and no extra memory protection is necessary.

```
void Asynch_Exchange(int proc_id) {
  np = chromosome_size/ number_of_proc;
  // Write phase
  Lock(&mutex);
  for (i=(id-1)*np; i<id*np; i++)
    best_chrom[i] =
               best_partial_chr[id][i];
  Unlock(&mutex);
  // Read phase
  Lock(&mutex);
  for (i=0; i<(id-1)*np; i++)
    template[id][i] = best_chrom[i];
  for (i=id*np; i<n; i++)
    template[id][i] = best_chrom[i];
  Unlock(&mutex);
}
```

Figure 4: Asynchronous exchange procedure for the best chromosome

In this version of the exchange procedure, the Barrier has been removed, and instead the memory is protected by a common mutual exclusion semaphore both in the reading phase and in the writing phase. The lock can be applied either to each read and write operation individually, or to the entire block of read or write operations for each process. The model shown here has the disadvantage that only one process can read or write at a time. This could be optimized by having a dedicated semaphore for each portion of the chromosome belonging to each process. Given the fact that the exchange does not happen very often, and the complexity of our fitness function, this detail would not impact the timing in any visible way.

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

3. Motorcycle Driving Problem

We have chosen a complex problem to test our parallel model where the evaluation of the fitness is computationally expensive and requires a non-uniform amount of time. For this kind of problem we need some additional steps to measure the speedup accurately in the multi-core environment.

The problem consists in optimizing the parameters defining a pilot for a simulated motorcycle. For this problem, the evaluation requires significantly more computations than the genetic operators, and thus it will allow us to observe the improvement in performance in that respect. Furthermore, 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.

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

3.1 Simulated Motorcycle

The physical model of the motorcycle has been more extensively described in (Vrajitoru and Mehler 2005) and is close to (Getz 1994). The motorcycle or STV (single-track vehicle), is modeled as a system composed of several elements with various degrees of freedom that can be driven through several control units. Figure 5 shows the components of the physical model for a motorcycle.

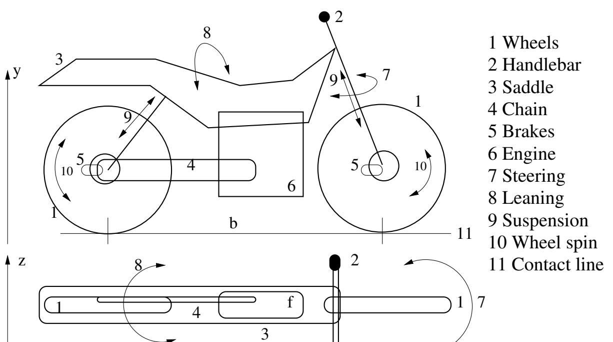

Figure 5: A motorcycle with control units and degrees of freedom

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. A human or artificial 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

$$q = (s, \alpha, \theta)^T \tag{1}$$

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

$$s(t + \Delta t) = s(t) + \Delta s, v(t + \Delta t) = v(t) + \Delta v \tag{2}$$

where:

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

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.

3.2 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 $front$, is 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, quantized 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 6 shows an example of the geometrical definition of these measures.

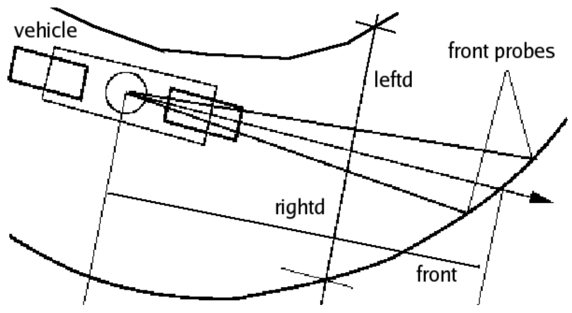

Figure 6: Perceptual information used by the autonomous pilot

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 (Vrajitoru and Mehler 2005). 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 can be 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. We use the combined learning/steering approach for this experiment.

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.

These agents are defined within the context of the motorcycle pilot and they are not directly related to the division of the chromosome between the threads or processes. For the research presented in this paper, the total number of parameters to be optimized and thus represented in a chromosome is such that a perfect distribution of these variables among the processes is possible. Thus, each variable belongs entirely to one single process.

However, the same thing cannot be said about the pilot agents because each of them has a different number of parameters it requires to function. The division into processes does not take this aspect into consideration, but it can definitely be considered for future research.

3.3 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 360.

We used the one-point crossover for our experiments with a probability of 0.8 and a probability of mutation of 0.01.

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
X4	Intel Quad Core Q6600	2.4 GHz	4	XP

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 fitness we marked 50 reference points on the road and counted how many of them the motorcycle passed by closely enough. The fitness is computed as follows:

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

where d_m is the number of points crossed by the motorcycle, d_t is the total number of reference 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.

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 execution time/speedup and the fitness performance.

Table 1 introduces the three platforms that we 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. Thus, one platform is single-cored, one is double-cored, and the third one is a QuadCore.

Table 2 shows the average execution time in number of seconds over 10 runs for both the synchronous (S in the second column) and the asynchronous (A in the second column) models. 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 10 runs. The row with 1 process represents the sequential GA used as a baseline. Table 3 shows the number of such calls divided by 10^4 as an average over 10 runs.

To compute the speedup for this problem, we need to compute a baseline of comparison that does not depend on

Table 2: Timing in seconds, 1000 generations, average over 10 runs

#Threads	Comm	Platform		
		U1	M2	X4
1		1969.3	1789.7	2081.0
2	S	6488.0	7459.4	3054.7
2	A	11985.9	4381.5	5638.4
4	S	13308.1	8396.22	5452.5
4	A	7140.5	8058.3	5156.4
8	S	32031.9	12882.0	16955.7
8	A	42254.5	5809.2	13561.6

Table 3: Number of moves divided by 10^4, 1000 generations, average over 10 runs

#Threads	Comm	Platform		
		U1	M2	X4
1		18678.0	1881.98	4051.34
2	S	4180.98	6578.59	7763.57
2	A	7080.79	3732.11	12176.10
4	S	5849.64	7344.52	5942.87
4	A	5009.33	7026.47	5037.21
8	S	11063.75	13074.47	14961.18
8	A	14515.89	10269.06	10394.82

Table 4: Computational time for every 10^4 moves

#Threads	Comm	Platform		
		U1	M2	X4
1		1.05	0.95	0.51
2	S	1.55	1.13	0.39
2	A	1.69	1.17	0.46
4	S	2.28	1.14	0.92
4	A	1.43	1.15	1.02
8	S	2.90	0.99	1.13
8	A	2.91	0.57	1.30

Table 5: Speedup in 1000 generations

#Threads	Comm	Platform		
		U1	M2	X4
2	S	67.94%	83.87%	130.55%
2	A	62.28%	81.00%	110.92%
4	S	46.34%	83.19%	55.99%
4	A	73.96%	82.92%	50.18%
8	S	36.41%	96.52%	45.32%
8	A	36.22%	168.10%	39.37%

the number of operations executed for each case. Thus, Table 4 computes the average time is seconds for every 10^4 move operations by dividing the time in Table 2 by the number of moves in Table 3. Then Table 5 shows the speedup for this problem by dividing this new timing measure for the sequential case by its value for the parallel case. Higher values for the speedup are preferable and values over 100% show that the parallel version of the algorithm is faster than the sequential one.

Finally, Table 6 shows the average and maximum fitness achieved in 1000 generations over the 10 runs as an average over the 3 platforms because the fitness is platform-independent, at least in theory. Higher numbers are better, and a fitness value over 1.0 is an indication that the motorcycle pilot might have completed the circuit in that case. This table shows that both parallel models are an improvement over the sequential one for two processes. For higher numbers of processes there is a significant improvement for the asynchronous model over both the synchronous model and over the sequential program. This validates our parallel approach.

5. Conclusions

In this paper we have presented a multi-threaded genetic algorithm dividing the problem and the genes of the chromosome among the processes instead of dividing the population. The model is designed for multi-core processor archi-

Table 6: Average fitness after 1000 generations over the 3 platforms

Comm	#Threads			
	1	2	4	8
Ave S	0.9213	0.9793	0.8578	0.8886
Max S	1.0000	1.0000	1.1080	1.2000
Ave A	0.9213	0.9682	0.9601	0.9378
Max A	1.0000	1.0000	1.0000	1.3334

tectures. The focus of this paper is on a feature of the algorithm related to communication. We have tried to see how a lack of synchronization of the threads along the generations affect the execution time and the fitness.

Section 2 introduced our parallel model and explained the difference between the synchronized model and the asynchronous one. Section 3 introduced the problem used for testing the model. Section 4 presented our experimental results designed to answer our question.

The experimental results indicate that the synchronous version of the genetic algorithm is more efficient than the asynchronous one in terms of speedup, which is a rather surprising result. By looking at the speedup by platform, there is a clear improvement for the 4 core processor platform with 2 threads, while on the same platform the speedup for 4 threads does not seem to follow a similar pattern. This could indicate that the pthread library or the Windows XP operating system are not yet optimized to take advantage of the QuadCore architecture. For future research we will repeat the experiment on the same platform with the Windows 7 operating system.

In terms of fitness we can see a clear improvement from the sequential model to both of the parallel ones and a better performance of the asynchronous model overall, both for the average and for the maximum fitness.

Overall the parallel model is a good way to take advantage of the extra computational power of these architectures while improving the performance of the algorithm.

References

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

Baker, B.; Carter, C.; and Dozier, G. 2009. Sema: A new paradigm for distributed genetic and evolutionary computating. In Kim, B., ed., *Proceeding of the Midwest Artificial Intelligence and Cognitive Science Conference*, 35–39.

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

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

Dozier, G. 2003. 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.

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

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

Kim, M.; Aggarwal, V.; O'Reilly, U.; and Médard, M. 2007. A doubly distributed genetic algorithm for network coding. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1272–1279. London, UK: ACM.

Llorà, X.; Reddy, R.; Matesic, B.; and Bhargava, R. 2007. Towards better than human capability in diagnosing prostate cancer using infrared spectroscopic imaging. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2098–2105. London, UK: ACM.

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

Sekaj, I., and Oravec, M. 2009. 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*, 945–948. New York, NY, USA: ACM.

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

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

Z. Skolicki, K. D. J. 2005. The influence of migration sizes and intervals on island models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1295–1302.